Python教程II

-----(涉及html,web,应用等)

#### 常用内建模块

Python之所以自称“batteries included”，就是因为内置了许多非常有用的模块，无需额外安装和配置，即可直接使用。

本章将介绍一些常用的内建模块。

### 感觉本站内容不错，读后有收获？

#### datetime

阅读: 65370

datetime是Python处理日期和时间的标准库。

### 获取当前日期和时间

我们先看如何获取当前日期和时间：

>>> from datetime import datetime

>>> now = datetime.now() *# 获取当前datetime*

>>> **print**(now)

2015-05-18 16:28:07.198690

>>> **print**(type(now))

<**class 'datetime.datetime'>**

注意到datetime是模块，datetime模块还包含一个datetime类，通过from datetime import datetime导入的才是datetime这个类。

如果仅导入import datetime，则必须引用全名datetime.datetime。

datetime.now()返回当前日期和时间，其类型是datetime。

### 获取指定日期和时间

要指定某个日期和时间，我们直接用参数构造一个datetime：

>>> **from** datetime **import** datetime

>>> dt = datetime(2015, 4, 19, 12, 20) *# 用指定日期时间创建datetime*

>>> print(dt)

2015-04-19 12:20:00

### datetime转换为timestamp

在计算机中，时间实际上是用数字表示的。我们把1970年1月1日 00:00:00 UTC+00:00时区的时刻称为epoch time，记为0（1970年以前的时间timestamp为负数），当前时间就是相对于epoch time的秒数，称为timestamp。

你可以认为：

timestamp = 0 = 1970-1-1 00:00:00 UTC+0:00

对应的北京时间是：

timestamp = 0 = 1970-1-1 08:00:00 UTC+8:00

可见timestamp的值与时区毫无关系，因为timestamp一旦确定，其UTC时间就确定了，转换到任意时区的时间也是完全确定的，这就是为什么计算机存储的当前时间是以timestamp表示的，因为全球各地的计算机在任意时刻的timestamp都是完全相同的（假定时间已校准）。

把一个datetime类型转换为timestamp只需要简单调用timestamp()方法：

>>> **from** datetime **import** datetime

>>> dt = datetime(2015, 4, 19, 12, 20) *# 用指定日期时间创建datetime*

>>> dt.timestamp() *# 把datetime转换为timestamp*

1429417200.0

注意Python的timestamp是一个浮点数。如果有小数位，小数位表示毫秒数。

某些编程语言（如Java和JavaScript）的timestamp使用整数表示毫秒数，这种情况下只需要把timestamp除以1000就得到Python的浮点表示方法。

### timestamp转换为datetime

要把timestamp转换为datetime，使用datetime提供的fromtimestamp()方法：

>>> **from** datetime **import** datetime

>>> t = 1429417200.0

>>> print(datetime.fromtimestamp(t))

2015-04-19 12:20:00

注意到timestamp是一个浮点数，它没有时区的概念，而datetime是有时区的。上述转换是在timestamp和本地时间做转换。

本地时间是指当前操作系统设定的时区。例如北京时区是东8区，则本地时间：

2015-04-19 12:20:00

实际上就是UTC+8:00时区的时间：

2015-04-19 12:20:00 UTC+8:00

而此刻的格林威治标准时间与北京时间差了8小时，也就是UTC+0:00时区的时间应该是：

2015-04-19 04:20:00 UTC+0:00

timestamp也可以直接被转换到UTC标准时区的时间：

>>> **from** datetime **import** datetime

>>> t = 1429417200.0

>>> print(datetime.fromtimestamp(t)) *# 本地时间*

2015-04-19 12:20:00

>>> print(datetime.utcfromtimestamp(t)) *# UTC时间*

2015-04-19 04:20:00

### str转换为datetime

很多时候，用户输入的日期和时间是字符串，要处理日期和时间，首先必须把str转换为datetime。转换方法是通过datetime.strptime()实现，需要一个日期和时间的格式化字符串：

>>> **from** datetime **import** datetime

>>> cday = datetime.strptime('2015-6-1 18:19:59', '%Y-%m-%d %H:%M:%S')

>>> print(cday)

2015-06-01 18:19:59

字符串'%Y-%m-%d %H:%M:%S'规定了日期和时间部分的格式。详细的说明请参考[Python文档](https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior)。

注意转换后的datetime是没有时区信息的。

### datetime转换为str

如果已经有了datetime对象，要把它格式化为字符串显示给用户，就需要转换为str，转换方法是通过strftime()实现的，同样需要一个日期和时间的格式化字符串：

>>> **from** datetime **import** datetime

>>> now = datetime.now()

>>> print(now.strftime('%a, %b %d %H:%M'))

Mon, May 05 16:28

### datetime加减

对日期和时间进行加减实际上就是把datetime往后或往前计算，得到新的datetime。加减可以直接用+和-运算符，不过需要导入timedelta这个类：

>>> **from** datetime **import** datetime, timedelta

>>> now = datetime.now()

>>> now

datetime.datetime(2015, 5, 18, 16, 57, 3, 540997)

>>> now + timedelta(hours=10)

datetime.datetime(2015, 5, 19, 2, 57, 3, 540997)

>>> now - timedelta(days=1)

datetime.datetime(2015, 5, 17, 16, 57, 3, 540997)

>>> now + timedelta(days=2, hours=12)

datetime.datetime(2015, 5, 21, 4, 57, 3, 540997)

可见，使用timedelta你可以很容易地算出前几天和后几天的时刻。

### 本地时间转换为UTC时间

本地时间是指系统设定时区的时间，例如北京时间是UTC+8:00时区的时间，而UTC时间指UTC+0:00时区的时间。

一个datetime类型有一个时区属性tzinfo，但是默认为None，所以无法区分这个datetime到底是哪个时区，除非强行给datetime设置一个时区：

>>> **from** datetime **import** datetime, timedelta, timezone

>>> tz\_utc\_8 = timezone(timedelta(hours=8)) *# 创建时区UTC+8:00*

>>> now = datetime.now()

>>> now

datetime.datetime(2015, 5, 18, 17, 2, 10, 871012)

>>> dt = now.replace(tzinfo=tz\_utc\_8) *# 强制设置为UTC+8:00*

>>> dt

datetime.datetime(2015, 5, 18, 17, 2, 10, 871012, tzinfo=datetime.timezone(datetime.timedelta(0, 28800)))

如果系统时区恰好是UTC+8:00，那么上述代码就是正确的，否则，不能强制设置为UTC+8:00时区。

### 时区转换

我们可以先通过utcnow()拿到当前的UTC时间，再转换为任意时区的时间：

*# 拿到UTC时间，并强制设置时区为UTC+0:00:*

>>> utc\_dt = datetime.utcnow().replace(tzinfo=timezone.utc)

>>> print(utc\_dt)

2015-05-18 09:05:12.377316+00:00

*# astimezone()将转换时区为北京时间:*

>>> bj\_dt = utc\_dt.astimezone(timezone(timedelta(hours=8)))

>>> print(bj\_dt)

2015-05-18 17:05:12.377316+08:00

*# astimezone()将转换时区为东京时间:*

>>> tokyo\_dt = utc\_dt.astimezone(timezone(timedelta(hours=9)))

>>> print(tokyo\_dt)

2015-05-18 18:05:12.377316+09:00

*# astimezone()将bj\_dt转换时区为东京时间:*

>>> tokyo\_dt2 = bj\_dt.astimezone(timezone(timedelta(hours=9)))

>>> print(tokyo\_dt2)

2015-05-18 18:05:12.377316+09:00

时区转换的关键在于，拿到一个datetime时，要获知其正确的时区，然后强制设置时区，作为基准时间。

利用带时区的datetime，通过astimezone()方法，可以转换到任意时区。

注：不是必须从UTC+0:00时区转换到其他时区，任何带时区的datetime都可以正确转换，例如上述bj\_dt到tokyo\_dt的转换。

### 小结

datetime表示的时间需要时区信息才能确定一个特定的时间，否则只能视为本地时间。

如果要存储datetime，最佳方法是将其转换为timestamp再存储，因为timestamp的值与时区完全无关。

### 练习

假设你获取了用户输入的日期和时间如2015-1-21 9:01:30，以及一个时区信息如UTC+5:00，均是str，请编写一个函数将其转换为timestamp：

窗体顶端

# -\*- coding:utf-8 -\*-

import re

from datetime import datetime, timezone, timedelta

def to\_timestamp(dt\_str, tz\_str):



# 测试:

t1 = to\_timestamp('2015-6-1 08:10:30', 'UTC+7:00')

assert t1 == 1433121030.0, t1

t2 = to\_timestamp('2015-5-31 16:10:30', 'UTC-09:00')

assert t2 == 1433121030.0, t2

print('Pass')

窗体底端

### 参考源码

[use\_datetime.py](https://github.com/michaelliao/learn-python3/blob/master/samples/commonlib/use_datetime.py)

### 感觉本站内容不错，读后有收获？

#### collections

阅读: 34289

collections是Python内建的一个集合模块，提供了许多有用的集合类。

### namedtuple

我们知道tuple可以表示不变集合，例如，一个点的二维坐标就可以表示成：

>>> p = (1, 2)

但是，看到(1, 2)，很难看出这个tuple是用来表示一个坐标的。

定义一个class又小题大做了，这时，namedtuple就派上了用场：

>>> **from** collections **import** namedtuple

>>> Point = namedtuple('Point', ['x', 'y'])

>>> p = Point(1, 2)

>>> p.x

1

>>> p.y

2

namedtuple是一个函数，它用来创建一个自定义的tuple对象，并且规定了tuple元素的个数，并可以用属性而不是索引来引用tuple的某个元素。

这样一来，我们用namedtuple可以很方便地定义一种数据类型，它具备tuple的不变性，又可以根据属性来引用，使用十分方便。

可以验证创建的Point对象是tuple的一种子类：

>>> isinstance(p, Point)

True

>>> isinstance(p, tuple)

True

类似的，如果要用坐标和半径表示一个圆，也可以用namedtuple定义：

*# namedtuple('名称', [属性list]):*

Circle = namedtuple('Circle', ['x', 'y', 'r'])

### deque

使用list存储数据时，按索引访问元素很快，但是插入和删除元素就很慢了，因为list是线性存储，数据量大的时候，插入和删除效率很低。

deque是为了高效实现插入和删除操作的双向列表，适合用于队列和栈：

>>> **from** collections **import** deque

>>> q = deque(['a', 'b', 'c'])

>>> q.append('x')

>>> q.appendleft('y')

>>> q

deque(['y', 'a', 'b', 'c', 'x'])

deque除了实现list的append()和pop()外，还支持appendleft()和popleft()，这样就可以非常高效地往头部添加或删除元素。

### defaultdict

使用dict时，如果引用的Key不存在，就会抛出KeyError。如果希望key不存在时，返回一个默认值，就可以用defaultdict：

>>> **from** collections **import** defaultdict

>>> dd = defaultdict(**lambda**: 'N/A')

>>> dd['key1'] = 'abc'

>>> dd['key1'] *# key1存在*

'abc'

>>> dd['key2'] *# key2不存在，返回默认值*

'N/A'

注意默认值是调用函数返回的，而函数在创建defaultdict对象时传入。

除了在Key不存在时返回默认值，defaultdict的其他行为跟dict是完全一样的。

### OrderedDict

使用dict时，Key是无序的。在对dict做迭代时，我们无法确定Key的顺序。

如果要保持Key的顺序，可以用OrderedDict：

>>> **from** collections **import** OrderedDict

>>> d = dict([('a', 1), ('b', 2), ('c', 3)])

>>> d *# dict的Key是无序的*

{'a': 1, 'c': 3, 'b': 2}

>>> od = OrderedDict([('a', 1), ('b', 2), ('c', 3)])

>>> od *# OrderedDict的Key是有序的*

OrderedDict([('a', 1), ('b', 2), ('c', 3)])

注意，OrderedDict的Key会按照插入的顺序排列，不是Key本身排序：

>>> od = OrderedDict()

>>> od['z'] = 1

>>> od['y'] = 2

>>> od['x'] = 3

>>> list(od.keys()) *# 按照插入的Key的顺序返回*

['z', 'y', 'x']

OrderedDict可以实现一个FIFO（先进先出）的dict，当容量超出限制时，先删除最早添加的Key：

**from** collections **import** OrderedDict

**class LastUpdatedOrderedDict(OrderedDict):**

**def** **\_\_init\_\_**(self, capacity):

super(LastUpdatedOrderedDict, self).\_\_init\_\_()

self.\_capacity = capacity

**def** **\_\_setitem\_\_**(self, key, value):

containsKey = 1 **if** key **in** self **else** 0

**if** len(self) - containsKey >= self.\_capacity:

last = self.popitem(last=False)

print('remove:', last)

**if** containsKey:

**del** self[key]

print('set:', (key, value))

**else**:

print('add:', (key, value))

OrderedDict.\_\_setitem\_\_(self, key, value)

### Counter

Counter是一个简单的计数器，例如，统计字符出现的个数：

>>> **from** collections **import** Counter

>>> c = Counter()

>>> **for** ch **in** 'programming':

... c[ch] = c[ch] + 1

...

>>> c

Counter({'g': 2, 'm': 2, 'r': 2, 'a': 1, 'i': 1, 'o': 1, 'n': 1, 'p': 1})

Counter实际上也是dict的一个子类，上面的结果可以看出，字符'g'、'm'、'r'各出现了两次，其他字符各出现了一次。

### 小结

collections模块提供了一些有用的集合类，可以根据需要选用。

### 参考源码

[use\_collections.py](https://github.com/michaelliao/learn-python3/blob/master/samples/commonlib/use_collections.py)

### 感觉本站内容不错，读后有收获？

#### base64

阅读: 21974

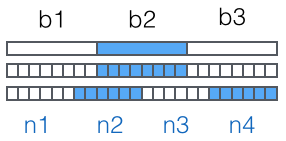
Base64是一种用64个字符来表示任意二进制数据的方法。

用记事本打开exe、jpg、pdf这些文件时，我们都会看到一大堆乱码，因为二进制文件包含很多无法显示和打印的字符，所以，如果要让记事本这样的文本处理软件能处理二进制数据，就需要一个二进制到字符串的转换方法。Base64是一种最常见的二进制编码方法。

Base64的原理很简单，首先，准备一个包含64个字符的数组：

['A', 'B', 'C', ... 'a', 'b', 'c', ... '0', '1', ... '+', '/']

然后，对二进制数据进行处理，每3个字节一组，一共是3x8=24bit，划为4组，每组正好6个bit：



这样我们得到4个数字作为索引，然后查表，获得相应的4个字符，就是编码后的字符串。

所以，Base64编码会把3字节的二进制数据编码为4字节的文本数据，长度增加33%，好处是编码后的文本数据可以在邮件正文、网页等直接显示。

如果要编码的二进制数据不是3的倍数，最后会剩下1个或2个字节怎么办？Base64用\x00字节在末尾补足后，再在编码的末尾加上1个或2个=号，表示补了多少字节，解码的时候，会自动去掉。

Python内置的base64可以直接进行base64的编解码：

>>> **import** base64

>>> base64.b64encode(b'binary\x00string')

b'YmluYXJ5AHN0cmluZw=='

>>> base64.b64decode(b'YmluYXJ5AHN0cmluZw==')

b'binary\x00string'

由于标准的Base64编码后可能出现字符+和/，在URL中就不能直接作为参数，所以又有一种"url safe"的base64编码，其实就是把字符+和/分别变成-和\_：

>>> base64.b64encode(b'i\xb7\x1d\xfb\xef\xff')

b'abcd++//'

>>> base64.urlsafe\_b64encode(b'i\xb7\x1d\xfb\xef\xff')

b'abcd--\_\_'

>>> base64.urlsafe\_b64decode('abcd--\_\_')

b'i\xb7\x1d\xfb\xef\xff'

还可以自己定义64个字符的排列顺序，这样就可以自定义Base64编码，不过，通常情况下完全没有必要。

Base64是一种通过查表的编码方法，不能用于加密，即使使用自定义的编码表也不行。

Base64适用于小段内容的编码，比如数字证书签名、Cookie的内容等。

由于=字符也可能出现在Base64编码中，但=用在URL、Cookie里面会造成歧义，所以，很多Base64编码后会把=去掉：

*# 标准Base64:*

'abcd' -> 'YWJjZA=='

*# 自动去掉=:*

'abcd' -> 'YWJjZA'

去掉=后怎么解码呢？因为Base64是把3个字节变为4个字节，所以，Base64编码的长度永远是4的倍数，因此，需要加上=把Base64字符串的长度变为4的倍数，就可以正常解码了。

### 小结

Base64是一种任意二进制到文本字符串的编码方法，常用于在URL、Cookie、网页中传输少量二进制数据。

### 练习

请写一个能处理去掉=的base64解码函数：

窗体顶端

# -\*- coding: utf-8 -\*-

import base64

def safe\_base64\_decode(s):



# 测试:

assert b'abcd' == safe\_base64\_decode(b'YWJjZA=='), safe\_base64\_decode('YWJjZA==')

assert b'abcd' == safe\_base64\_decode(b'YWJjZA'), safe\_base64\_decode('YWJjZA')

print('Pass')

窗体底端

### 参考源码

[do\_base64.py](https://github.com/michaelliao/learn-python3/blob/master/samples/commonlib/do_base64.py)

### 感觉本站内容不错，读后有收获？

#### struct

阅读: 18117

准确地讲，Python没有专门处理字节的数据类型。但由于b'str'可以表示字节，所以，字节数组＝二进制str。而在C语言中，我们可以很方便地用struct、union来处理字节，以及字节和int，float的转换。

在Python中，比方说要把一个32位无符号整数变成字节，也就是4个长度的bytes，你得配合位运算符这么写：

>>> n = 10240099

>>> b1 = (n & 0xff000000) >> 24

>>> b2 = (n & 0xff0000) >> 16

>>> b3 = (n & 0xff00) >> 8

>>> b4 = n & 0xff

>>> bs = bytes([b1, b2, b3, b4])

>>> bs

b'\x00\x9c@c'

非常麻烦。如果换成浮点数就无能为力了。

好在Python提供了一个struct模块来解决bytes和其他二进制数据类型的转换。

struct的pack函数把任意数据类型变成bytes：

>>> **import** struct

>>> struct.pack('>I', 10240099)

b'\x00\x9c@c'

pack的第一个参数是处理指令，'>I'的意思是：

>表示字节顺序是big-endian，也就是网络序，I表示4字节无符号整数。

后面的参数个数要和处理指令一致。

unpack把bytes变成相应的数据类型：

>>> struct.unpack('>IH', b'\xf0\xf0\xf0\xf0\x80\x80')

(4042322160, 32896)

根据>IH的说明，后面的bytes依次变为I：4字节无符号整数和H：2字节无符号整数。

所以，尽管Python不适合编写底层操作字节流的代码，但在对性能要求不高的地方，利用struct就方便多了。

struct模块定义的数据类型可以参考Python官方文档：

<https://docs.python.org/3/library/struct.html#format-characters>

Windows的位图文件（.bmp）是一种非常简单的文件格式，我们来用struct分析一下。

首先找一个bmp文件，没有的话用“画图”画一个。

读入前30个字节来分析：

>>> s = b'\x42\x4d\x38\x8c\x0a\x00\x00\x00\x00\x00\x36\x00\x00\x00\x28\x00\x00\x00\x80\x02\x00\x00\x68\x01\x00\x00\x01\x00\x18\x00'

BMP格式采用小端方式存储数据，文件头的结构按顺序如下：

两个字节：'BM'表示Windows位图，'BA'表示OS/2位图； 一个4字节整数：表示位图大小； 一个4字节整数：保留位，始终为0； 一个4字节整数：实际图像的偏移量； 一个4字节整数：Header的字节数； 一个4字节整数：图像宽度； 一个4字节整数：图像高度； 一个2字节整数：始终为1； 一个2字节整数：颜色数。

所以，组合起来用unpack读取：

>>> struct.unpack('<ccIIIIIIHH', s)

(b'B', b'M', 691256, 0, 54, 40, 640, 360, 1, 24)

结果显示，b'B'、b'M'说明是Windows位图，位图大小为640x360，颜色数为24。

请编写一个bmpinfo.py，可以检查任意文件是否是位图文件，如果是，打印出图片大小和颜色数。

### 参考源码

[check\_bmp.py](https://github.com/michaelliao/learn-python3/blob/master/samples/commonlib/check_bmp.py)

### 感觉本站内容不错，读后有收获？

#### hashlib

阅读: 15158

### 摘要算法简介

Python的hashlib提供了常见的摘要算法，如MD5，SHA1等等。

什么是摘要算法呢？摘要算法又称哈希算法、散列算法。它通过一个函数，把任意长度的数据转换为一个长度固定的数据串（通常用16进制的字符串表示）。

举个例子，你写了一篇文章，内容是一个字符串'how to use python hashlib - by Michael'，并附上这篇文章的摘要是'2d73d4f15c0db7f5ecb321b6a65e5d6d'。如果有人篡改了你的文章，并发表为'how to use python hashlib - by Bob'，你可以一下子指出Bob篡改了你的文章，因为根据'how to use python hashlib - by Bob'计算出的摘要不同于原始文章的摘要。

可见，摘要算法就是通过摘要函数f()对任意长度的数据data计算出固定长度的摘要digest，目的是为了发现原始数据是否被人篡改过。

摘要算法之所以能指出数据是否被篡改过，就是因为摘要函数是一个单向函数，计算f(data)很容易，但通过digest反推data却非常困难。而且，对原始数据做一个bit的修改，都会导致计算出的摘要完全不同。

我们以常见的摘要算法MD5为例，计算出一个字符串的MD5值：

**import** hashlib

md5 = hashlib.md5()

md5.update('how to use md5 in python hashlib?'.encode('utf-8'))

print(md5.hexdigest())

计算结果如下：

d26a53750bc40b38b65a520292f69306

如果数据量很大，可以分块多次调用update()，最后计算的结果是一样的：

import hashlib

md5 = hashlib.md5()

md5.**update**('how to use md5 in '.encode('utf-8'))

md5.**update**('python hashlib?'.encode('utf-8'))

print(md5.hexdigest())

试试改动一个字母，看看计算的结果是否完全不同。

MD5是最常见的摘要算法，速度很快，生成结果是固定的128 bit字节，通常用一个32位的16进制字符串表示。

另一种常见的摘要算法是SHA1，调用SHA1和调用MD5完全类似：

import hashlib

sha1 = hashlib.sha1()

sha1.**update**('how to use sha1 in '.encode('utf-8'))

sha1.**update**('python hashlib?'.encode('utf-8'))

print(sha1.hexdigest())

SHA1的结果是160 bit字节，通常用一个40位的16进制字符串表示。

比SHA1更安全的算法是SHA256和SHA512，不过越安全的算法不仅越慢，而且摘要长度更长。

有没有可能两个不同的数据通过某个摘要算法得到了相同的摘要？完全有可能，因为任何摘要算法都是把无限多的数据集合映射到一个有限的集合中。这种情况称为碰撞，比如Bob试图根据你的摘要反推出一篇文章'how to learn hashlib in python - by Bob'，并且这篇文章的摘要恰好和你的文章完全一致，这种情况也并非不可能出现，但是非常非常困难。

### 摘要算法应用

摘要算法能应用到什么地方？举个常用例子：

任何允许用户登录的网站都会存储用户登录的用户名和口令。如何存储用户名和口令呢？方法是存到数据库表中：

name | password

*--------+----------*

michael | 123456

bob | abc999

alice | alice2008

如果以明文保存用户口令，如果数据库泄露，所有用户的口令就落入黑客的手里。此外，网站运维人员是可以访问数据库的，也就是能获取到所有用户的口令。

正确的保存口令的方式是不存储用户的明文口令，而是存储用户口令的摘要，比如MD5：

username | password

*---------+---------------------------------*

michael | e10adc3949ba59abbe56e057f20f883e

bob | 878ef96e86145580c38c87f0410ad153

alice | 99b1c2188db85afee403b1536010c2c9

当用户登录时，首先计算用户输入的明文口令的MD5，然后和数据库存储的MD5对比，如果一致，说明口令输入正确，如果不一致，口令肯定错误。

### 练习

根据用户输入的口令，计算出存储在数据库中的MD5口令：

**def** **calc\_md5**(password):

**pass**

存储MD5的好处是即使运维人员能访问数据库，也无法获知用户的明文口令。

设计一个验证用户登录的函数，根据用户输入的口令是否正确，返回True或False：

db = {

'michael': 'e10adc3949ba59abbe56e057f20f883e',

'bob': '878ef96e86145580c38c87f0410ad153',

'alice': '99b1c2188db85afee403b1536010c2c9'

}

**def** **login**(user, password):

**pass**

采用MD5存储口令是否就一定安全呢？也不一定。假设你是一个黑客，已经拿到了存储MD5口令的数据库，如何通过MD5反推用户的明文口令呢？暴力破解费事费力，真正的黑客不会这么干。

考虑这么个情况，很多用户喜欢用123456，888888，password这些简单的口令，于是，黑客可以事先计算出这些常用口令的MD5值，得到一个反推表：

'e10adc3949ba59abbe56e057f20f883e': '123456'

'21218cca77804d2ba1922c33e0151105': '888888'

'5f4dcc3b5aa765d61d8327deb882cf99': 'password'

这样，无需破解，只需要对比数据库的MD5，黑客就获得了使用常用口令的用户账号。

对于用户来讲，当然不要使用过于简单的口令。但是，我们能否在程序设计上对简单口令加强保护呢？

由于常用口令的MD5值很容易被计算出来，所以，要确保存储的用户口令不是那些已经被计算出来的常用口令的MD5，这一方法通过对原始口令加一个复杂字符串来实现，俗称“加盐”：

**def** **calc\_md5**(password):

**return** get\_md5(password + 'the-Salt')

经过Salt处理的MD5口令，只要Salt不被黑客知道，即使用户输入简单口令，也很难通过MD5反推明文口令。

但是如果有两个用户都使用了相同的简单口令比如123456，在数据库中，将存储两条相同的MD5值，这说明这两个用户的口令是一样的。有没有办法让使用相同口令的用户存储不同的MD5呢？

如果假定用户无法修改登录名，就可以通过把登录名作为Salt的一部分来计算MD5，从而实现相同口令的用户也存储不同的MD5。

### 练习

根据用户输入的登录名和口令模拟用户注册，计算更安全的MD5：

db = {}

**def** **register**(username, password):

db[username] = get\_md5(password + username + 'the-Salt')

然后，根据修改后的MD5算法实现用户登录的验证：

**def** **login**(username, password):

**pass**

### 小结

摘要算法在很多地方都有广泛的应用。要注意摘要算法不是加密算法，不能用于加密（因为无法通过摘要反推明文），只能用于防篡改，但是它的单向计算特性决定了可以在不存储明文口令的情况下验证用户口令。

### 参考源码

[use\_hashlib.py](https://github.com/michaelliao/learn-python3/blob/master/samples/commonlib/use_hashlib.py)

### 感觉本站内容不错，读后有收获？

#### itertools

阅读: 12570

Python的内建模块itertools提供了非常有用的用于操作迭代对象的函数。

首先，我们看看itertools提供的几个“无限”迭代器：

>>> **import** itertools

>>> natuals = itertools.count(1)

>>> **for** n **in** natuals:

... print(n)

...

1

2

3

...

因为count()会创建一个无限的迭代器，所以上述代码会打印出自然数序列，根本停不下来，只能按Ctrl+C退出。

cycle()会把传入的一个序列无限重复下去：

>>> **import** itertools

>>> cs = itertools.cycle('ABC') *# 注意字符串也是序列的一种*

>>> **for** c **in** cs:

... print(c)

...

'A'

'B'

'C'

'A'

'B'

'C'

...

同样停不下来。

repeat()负责把一个元素无限重复下去，不过如果提供第二个参数就可以限定重复次数：

>>> ns = itertools.repeat('A', 3)

>>> **for** n **in** ns:

... print(n)

...

A

A

A

无限序列只有在for迭代时才会无限地迭代下去，如果只是创建了一个迭代对象，它不会事先把无限个元素生成出来，事实上也不可能在内存中创建无限多个元素。

无限序列虽然可以无限迭代下去，但是通常我们会通过takewhile()等函数根据条件判断来截取出一个有限的序列：

>>> natuals = itertools.count(1)

>>> ns = itertools.takewhile(**lambda** x: x <= 10, natuals)

>>> list(ns)

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

itertools提供的几个迭代器操作函数更加有用：

### chain()

chain()可以把一组迭代对象串联起来，形成一个更大的迭代器：

>>> **for** c **in** itertools.chain('ABC', 'XYZ'):

... print(c)

*# 迭代效果：'A' 'B' 'C' 'X' 'Y' 'Z'*

### groupby()

groupby()把迭代器中相邻的重复元素挑出来放在一起：

>>> **for** key, group **in** itertools.groupby('AAABBBCCAAA'):

... print(key, list(group))

...

A ['A', 'A', 'A']

B ['B', 'B', 'B']

C ['C', 'C']

A ['A', 'A', 'A']

实际上挑选规则是通过函数完成的，只要作用于函数的两个元素返回的值相等，这两个元素就被认为是在一组的，而函数返回值作为组的key。如果我们要忽略大小写分组，就可以让元素'A'和'a'都返回相同的key：

>>> **for** key, group **in** itertools.groupby('AaaBBbcCAAa', **lambda** c: c.upper()):

... print(key, list(group))

...

A ['A', 'a', 'a']

B ['B', 'B', 'b']

C ['c', 'C']

A ['A', 'A', 'a']

### 小结

itertools模块提供的全部是处理迭代功能的函数，它们的返回值不是list，而是Iterator，只有用for循环迭代的时候才真正计算。

### 参考源码

[use\_itertools.py](https://github.com/michaelliao/learn-python3/blob/master/samples/commonlib/use_itertools.py)

### 感觉本站内容不错，读后有收获？

#### XML

阅读: 25691

XML虽然比JSON复杂，在Web中应用也不如以前多了，不过仍有很多地方在用，所以，有必要了解如何操作XML。

### DOM vs SAX

操作XML有两种方法：DOM和SAX。DOM会把整个XML读入内存，解析为树，因此占用内存大，解析慢，优点是可以任意遍历树的节点。SAX是流模式，边读边解析，占用内存小，解析快，缺点是我们需要自己处理事件。

正常情况下，优先考虑SAX，因为DOM实在太占内存。

在Python中使用SAX解析XML非常简洁，通常我们关心的事件是start\_element，end\_element和char\_data，准备好这3个函数，然后就可以解析xml了。

举个例子，当SAX解析器读到一个节点时：

<a href="/">python</a>

会产生3个事件：

1. start\_element事件，在读取<a href="/">时；
2. char\_data事件，在读取python时；
3. end\_element事件，在读取</a>时。

用代码实验一下：

**from** xml.parsers.expat **import** ParserCreate

**class DefaultSaxHandler(object):**

**def** **start\_element**(self, name, attrs):

print('sax:start\_element: %s, attrs: %s' % (name, str(attrs)))

**def** **end\_element**(self, name):

print('sax:end\_element: %s' % name)

**def** **char\_data**(self, text):

print('sax:char\_data: %s' % text)

xml = r'''<?xml version="1.0"?>

<ol>

<li><a href="/python">Python</a></li>

<li><a href="/ruby">Ruby</a></li>

</ol>

'''

handler = DefaultSaxHandler()

parser = ParserCreate()

parser.StartElementHandler = handler.start\_element

parser.EndElementHandler = handler.end\_element

parser.CharacterDataHandler = handler.char\_data

parser.Parse(xml)

需要注意的是读取一大段字符串时，CharacterDataHandler可能被多次调用，所以需要自己保存起来，在EndElementHandler里面再合并。

除了解析XML外，如何生成XML呢？99%的情况下需要生成的XML结构都是非常简单的，因此，最简单也是最有效的生成XML的方法是拼接字符串：

L = []

L.append(r'<?xml version="1.0"?>')

L.append(r'<root>')

L.append(encode('some & data'))

L.append(r'</root>')

**return** ''.join(L)

如果要生成复杂的XML呢？建议你不要用XML，改成JSON。

### 小结

解析XML时，注意找出自己感兴趣的节点，响应事件时，把节点数据保存起来。解析完毕后，就可以处理数据。

### 练习

请利用SAX编写程序解析Yahoo的XML格式的天气预报，获取当天和第二天的天气：

<http://weather.yahooapis.com/forecastrss?u=c&w=2151330>

参数w是城市代码，要查询某个城市代码，可以在[weather.yahoo.com](https://weather.yahoo.com/)搜索城市，浏览器地址栏的URL就包含城市代码。

窗体顶端

# -\*- coding:utf-8 -\*-

from xml.parsers.expat import ParserCreate



# 测试:

data = r'''<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

<rss version="2.0" xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" xmlns:geo="http://www.w3.org/2003/01/geo/wgs84\_pos#">

<channel>

<title>Yahoo! Weather - Beijing, CN</title>

<lastBuildDate>Wed, 27 May 2015 11:00 am CST</lastBuildDate>

<yweather:location city="Beijing" region="" country="China"/>

<yweather:units temperature="C" distance="km" pressure="mb" speed="km/h"/>

<yweather:wind chill="28" direction="180" speed="14.48" />

<yweather:atmosphere humidity="53" visibility="2.61" pressure="1006.1" rising="0" />

<yweather:astronomy sunrise="4:51 am" sunset="7:32 pm"/>

<item>

<geo:lat>39.91</geo:lat>

<geo:long>116.39</geo:long>

<pubDate>Wed, 27 May 2015 11:00 am CST</pubDate>

<yweather:condition text="Haze" code="21" temp="28" date="Wed, 27 May 2015 11:00 am CST" />

<yweather:forecast day="Wed" date="27 May 2015" low="20" high="33" text="Partly Cloudy" code="30" />

<yweather:forecast day="Thu" date="28 May 2015" low="21" high="34" text="Sunny" code="32" />

<yweather:forecast day="Fri" date="29 May 2015" low="18" high="25" text="AM Showers" code="39" />

<yweather:forecast day="Sat" date="30 May 2015" low="18" high="32" text="Sunny" code="32" />

<yweather:forecast day="Sun" date="31 May 2015" low="20" high="37" text="Sunny" code="32" />

</item>

</channel>

</rss>

'''

weather = parse\_weather(data)

assert weather['city'] == 'Beijing', weather['city']

assert weather['country'] == 'China', weather['country']

assert weather['today']['text'] == 'Partly Cloudy', weather['today']['text']

assert weather['today']['low'] == 20, weather['today']['low']

assert weather['today']['high'] == 33, weather['today']['high']

assert weather['tomorrow']['text'] == 'Sunny', weather['tomorrow']['text']

assert weather['tomorrow']['low'] == 21, weather['tomorrow']['low']

assert weather['tomorrow']['high'] == 34, weather['tomorrow']['high']

print('Weather:', str(weather))

窗体底端

### 参考源码

[use\_sax.py](https://github.com/michaelliao/learn-python3/blob/master/samples/commonlib/use_sax.py)

### 感觉本站内容不错，读后有收获？

#### HTMLParser

阅读: 27013

如果我们要编写一个搜索引擎，第一步是用爬虫把目标网站的页面抓下来，第二步就是解析该HTML页面，看看里面的内容到底是新闻、图片还是视频。

假设第一步已经完成了，第二步应该如何解析HTML呢？

HTML本质上是XML的子集，但是HTML的语法没有XML那么严格，所以不能用标准的DOM或SAX来解析HTML。

好在Python提供了HTMLParser来非常方便地解析HTML，只需简单几行代码：

**from** html.parser **import** HTMLParser

**from** html.entities **import** name2codepoint

**class MyHTMLParser(HTMLParser):**

**def** **handle\_starttag**(self, tag, attrs):

print('<%s>' % tag)

**def** **handle\_endtag**(self, tag):

print('</%s>' % tag)

**def** **handle\_startendtag**(self, tag, attrs):

print('<%s/>' % tag)

**def** **handle\_data**(self, data):

print(data)

**def** **handle\_comment**(self, data):

print('<!--', data, '-->')

**def** **handle\_entityref**(self, name):

print('&%s;' % name)

**def** **handle\_charref**(self, name):

print('&#%s;' % name)

parser = MyHTMLParser()

parser.feed('''<html>

<head></head>

<body>

<!-- test html parser -->

<p>Some <a href=\"#\">html</a> HTML&nbsp;tutorial...<br>END</p>

</body></html>''')

feed()方法可以多次调用，也就是不一定一次把整个HTML字符串都塞进去，可以一部分一部分塞进去。

特殊字符有两种，一种是英文表示的&nbsp;，一种是数字表示的&#1234;，这两种字符都可以通过Parser解析出来。

### 小结

利用HTMLParser，可以把网页中的文本、图像等解析出来。

### 练习

找一个网页，例如<https://www.python.org/events/python-events/>，用浏览器查看源码并复制，然后尝试解析一下HTML，输出Python官网发布的会议时间、名称和地点。

### 参考源码

[use\_htmlparser.py](https://github.com/michaelliao/learn-python3/blob/master/samples/commonlib/use_htmlparser.py)

### 感觉本站内容不错，读后有收获？

#### urllib

阅读: 107296

urllib提供了一系列用于操作URL的功能。

### Get

urllib的request模块可以非常方便地抓取URL内容，也就是发送一个GET请求到指定的页面，然后返回HTTP的响应：

例如，对豆瓣的一个URLhttps://api.douban.com/v2/book/2129650进行抓取，并返回响应：

**from** urllib **import** request

**with** request.urlopen('https://api.douban.com/v2/book/2129650') **as** f:

data = f.read()

print('Status:', f.status, f.reason)

**for** k, v **in** f.getheaders():

print('%s: %s' % (k, v))

print('Data:', data.decode('utf-8'))

可以看到HTTP响应的头和JSON数据：

Status: 200 OK

Server: nginx

Date: Tue, 26 May 2015 10:02:27 GMT

Content-Type: application/json; char**set**=utf-8

Content-Length: 2049

**Connection**: **close**

Expires: Sun, 1 Jan 2006 01:00:00 GMT

**Pragma**: **no**-cache

Cache-Control: must-revalidate, **no**-cache, private

X-DAE-Node: pidl1

Data: {"rating":{"max":10,"numRaters":16,"average":"7.4","min":0},"subtitle":"","author":["廖雪峰编著"],"pubdate":"2007-6","tags":[{"count":20,"name":"spring","title":"spring"}...}

如果我们要想模拟浏览器发送GET请求，就需要使用Request对象，通过往Request对象添加HTTP头，我们就可以把请求伪装成浏览器。例如，模拟iPhone 6去请求豆瓣首页：

**from** urllib **import** request

req = request.Request('http://www.douban.com/')

req.add\_header('User-Agent', 'Mozilla/6.0 (iPhone; CPU iPhone OS 8\_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/10A5376e Safari/8536.25')

**with** request.urlopen(req) **as** f:

print('Status:', f.status, f.reason)

**for** k, v **in** f.getheaders():

print('%s: %s' % (k, v))

print('Data:', f.read().decode('utf-8'))

这样豆瓣会返回适合iPhone的移动版网页：

...

<meta name="viewport" content="width=device-width, user-scalable=no, initial-scale=1.0, minimum-scale=1.0, maximum-scale=1.0">

<meta name="format-detection" content="telephone=no">

<link rel="apple-touch-icon" sizes="57x57" href="http://img4.douban.com/pics/cardkit/launcher/57.png" />

...

### Post

如果要以POST发送一个请求，只需要把参数data以bytes形式传入。

我们模拟一个微博登录，先读取登录的邮箱和口令，然后按照weibo.cn的登录页的格式以username=xxx&password=xxx的编码传入：

**from** urllib **import** request, parse

print('Login to weibo.cn...')

email = input('Email: ')

passwd = input('Password: ')

login\_data = parse.urlencode([

('username', email),

('password', passwd),

('entry', 'mweibo'),

('client\_id', ''),

('savestate', '1'),

('ec', ''),

('pagerefer', 'https://passport.weibo.cn/signin/welcome?entry=mweibo&r=http%3A%2F%2Fm.weibo.cn%2F')

])

req = request.Request('https://passport.weibo.cn/sso/login')

req.add\_header('Origin', 'https://passport.weibo.cn')

req.add\_header('User-Agent', 'Mozilla/6.0 (iPhone; CPU iPhone OS 8\_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/10A5376e Safari/8536.25')

req.add\_header('Referer', 'https://passport.weibo.cn/signin/login?entry=mweibo&res=wel&wm=3349&r=http%3A%2F%2Fm.weibo.cn%2F')

**with** request.urlopen(req, data=login\_data.encode('utf-8')) **as** f:

print('Status:', f.status, f.reason)

**for** k, v **in** f.getheaders():

print('%s: %s' % (k, v))

print('Data:', f.read().decode('utf-8'))

如果登录成功，我们获得的响应如下：

Status: 200 OK

Server: nginx/1.2.0

...

**Set**-Cookie: SSOLoginState=1432620126; path=/; domain=weibo.cn

...

Data: {"retcode":20000000,"msg":"","data":{...,"uid":"1658384301"}}

如果登录失败，我们获得的响应如下：

...

Data: {"retcode":50011015,"msg":"\u7528\u6237\u540d\u6216\u5bc6\u7801\u9519\u8bef","data":{"username":"example@python.org","errline":536}}

### Handler

如果还需要更复杂的控制，比如通过一个Proxy去访问网站，我们需要利用ProxyHandler来处理，示例代码如下：

proxy\_**handler** = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})

proxy\_auth\_handler = urllib.request.ProxyBasicAuthHandler()

proxy\_auth\_handler.add\_password('realm', 'host', 'username', 'password')

opener = urllib.request.build\_opener(proxy\_handler, proxy\_auth\_handler)

**with** opener.**open**('http://www.example.com/login.html') **as** f:

pass

### 小结

urllib提供的功能就是利用程序去执行各种HTTP请求。如果要模拟浏览器完成特定功能，需要把请求伪装成浏览器。伪装的方法是先监控浏览器发出的请求，再根据浏览器的请求头来伪装，User-Agent头就是用来标识浏览器的。

### 练习

利用urllib读取XML，将XML一节的数据由硬编码改为由urllib获取：

窗体顶端

from urllib import request, parse

def fetch\_xml(url):



# 测试

print(fetch\_xml('http://weather.yahooapis.com/forecastrss?u=c&w=2151330'))

窗体底端

### 参考源码

[use\_urllib.py](https://github.com/michaelliao/learn-python3/blob/master/samples/commonlib/use_urllib.py)

### 感觉本站内容不错，读后有收获？

#### 常用第三方模块

阅读: 20748

除了内建的模块外，Python还有大量的第三方模块。

基本上，所有的第三方模块都会在[PyPI - the Python Package Index](https://pypi.python.org/)上注册，只要找到对应的模块名字，即可用pip安装。

本章介绍常用的第三方模块。

### 感觉本站内容不错，读后有收获？

#### PIL

阅读: 39511

PIL：Python Imaging Library，已经是Python平台事实上的图像处理标准库了。PIL功能非常强大，但API却非常简单易用。

由于PIL仅支持到Python 2.7，加上年久失修，于是一群志愿者在PIL的基础上创建了兼容的版本，名字叫[Pillow](https://github.com/python-pillow/Pillow)，支持最新Python 3.x，又加入了许多新特性，因此，我们可以直接安装使用Pillow。

### 安装Pillow

在命令行下直接通过pip安装：

$ pip install pillow

如果遇到Permission denied安装失败，请加上sudo重试。

### 操作图像

来看看最常见的图像缩放操作，只需三四行代码：

from PIL import Image

*# 打开一个jpg图像文件，注意是当前路径:*

im = Image.open('test.jpg')

*# 获得图像尺寸:*

w, h = im.size

**print**('Original image size: %sx%s' % (w, h))

*# 缩放到50%:*

im.thumbnail((w*//2, h//2))*

**print**('Resize image to: %sx%s' % (w*//2, h//2))*

*# 把缩放后的图像用jpeg格式保存:*

im.save('thumbnail.jpg', 'jpeg')

其他功能如切片、旋转、滤镜、输出文字、调色板等一应俱全。

比如，模糊效果也只需几行代码：

**from** PIL **import** Image, ImageFilter

*# 打开一个jpg图像文件，注意是当前路径:*

im = Image.open('test.jpg')

*# 应用模糊滤镜:*

im2 = im.filter(ImageFilter.BLUR)

im2.save('blur.jpg', 'jpeg')

效果如下：



PIL的ImageDraw提供了一系列绘图方法，让我们可以直接绘图。比如要生成字母验证码图片：

**from** PIL **import** Image, ImageDraw, ImageFont, ImageFilter

**import** random

*# 随机字母:*

**def** **rndChar**():

**return** chr(random.randint(65, 90))

*# 随机颜色1:*

**def** **rndColor**():

**return** (random.randint(64, 255), random.randint(64, 255), random.randint(64, 255))

*# 随机颜色2:*

**def** **rndColor2**():

**return** (random.randint(32, 127), random.randint(32, 127), random.randint(32, 127))

*# 240 x 60:*

width = 60 \* 4

height = 60

image = Image.new('RGB', (width, height), (255, 255, 255))

*# 创建Font对象:*

font = ImageFont.truetype('Arial.ttf', 36)

*# 创建Draw对象:*

draw = ImageDraw.Draw(image)

*# 填充每个像素:*

**for** x **in** range(width):

**for** y **in** range(height):

draw.point((x, y), fill=rndColor())

*# 输出文字:*

**for** t **in** range(4):

draw.text((60 \* t + 10, 10), rndChar(), font=font, fill=rndColor2())

*# 模糊:*

image = image.filter(ImageFilter.BLUR)

image.save('code.jpg', 'jpeg')

我们用随机颜色填充背景，再画上文字，最后对图像进行模糊，得到验证码图片如下：

验证码

如果运行的时候报错：

IOError: cannot open resource

这是因为PIL无法定位到字体文件的位置，可以根据操作系统提供绝对路径，比如：

'/Library/Fonts/Arial.ttf'

要详细了解PIL的强大功能，请请参考Pillow官方文档：

<https://pillow.readthedocs.org/>

### 小结

PIL提供了操作图像的强大功能，可以通过简单的代码完成复杂的图像处理。

### 参考源码

<https://github.com/michaelliao/learn-python3/blob/master/samples/packages/pil/use_pil_resize.py>

<https://github.com/michaelliao/learn-python3/blob/master/samples/packages/pil/use_pil_blur.py>

<https://github.com/michaelliao/learn-python3/blob/master/samples/packages/pil/use_pil_draw.py>

### 感觉本站内容不错，读后有收获？

#### virtualenv

阅读: 73348

在开发Python应用程序的时候，系统安装的Python3只有一个版本：3.4。所有第三方的包都会被pip安装到Python3的site-packages目录下。

如果我们要同时开发多个应用程序，那这些应用程序都会共用一个Python，就是安装在系统的Python 3。如果应用A需要jinja 2.7，而应用B需要jinja 2.6怎么办？

这种情况下，每个应用可能需要各自拥有一套“独立”的Python运行环境。virtualenv就是用来为一个应用创建一套“隔离”的Python运行环境。

首先，我们用pip安装virtualenv：

$ pip3 install virtualenv

然后，假定我们要开发一个新的项目，需要一套独立的Python运行环境，可以这么做：

第一步，创建目录：

Mac:~ michael$ mkdir myproject

Mac:~ michael$ cd myproject/

Mac:myproject michael$

第二步，创建一个独立的Python运行环境，命名为venv：

Mac:myproject michael$ virtualenv --no-site-packages venv

Using base prefix '/usr/local/.../Python.framework/Versions/3.4'

New python executable **in** venv/bin/python3.4

Also creating executable **in** venv/bin/python

Installing setuptools, pip, wheel...done.

命令virtualenv就可以创建一个独立的Python运行环境，我们还加上了参数--no-site-packages，这样，已经安装到系统Python环境中的所有第三方包都不会复制过来，这样，我们就得到了一个不带任何第三方包的“干净”的Python运行环境。

新建的Python环境被放到当前目录下的venv目录。有了venv这个Python环境，可以用source进入该环境：

Mac:myproject michael$ source venv/bin/activate

(venv)Mac:myproject michael$

注意到命令提示符变了，有个(venv)前缀，表示当前环境是一个名为venv的Python环境。

下面正常安装各种第三方包，并运行python命令：

(venv)Mac:myproject michael$ pip install jinja2

...

Successfully installed jinja2-2.7.3 markupsafe-0.23

(venv)Mac:myproject michael$ python myapp.py

...

在venv环境下，用pip安装的包都被安装到venv这个环境下，系统Python环境不受任何影响。也就是说，venv环境是专门针对myproject这个应用创建的。

退出当前的venv环境，使用deactivate命令：

(venv)Mac:myproject michael$ deactivate

Mac:myproject michael$

此时就回到了正常的环境，现在pip或python均是在系统Python环境下执行。

完全可以针对每个应用创建独立的Python运行环境，这样就可以对每个应用的Python环境进行隔离。

virtualenv是如何创建“独立”的Python运行环境的呢？原理很简单，就是把系统Python复制一份到virtualenv的环境，用命令source venv/bin/activate进入一个virtualenv环境时，virtualenv会修改相关环境变量，让命令python和pip均指向当前的virtualenv环境。

### 小结

virtualenv为应用提供了隔离的Python运行环境，解决了不同应用间多版本的冲突问题。

### 感觉本站内容不错，读后有收获？

#### 图形界面

阅读: 88753

Python支持多种图形界面的第三方库，包括：

* Tk
* wxWidgets
* Qt
* GTK

等等。

但是Python自带的库是支持Tk的Tkinter，使用Tkinter，无需安装任何包，就可以直接使用。本章简单介绍如何使用Tkinter进行GUI编程。

### Tkinter

我们来梳理一下概念：

我们编写的Python代码会调用内置的Tkinter，Tkinter封装了访问Tk的接口；

Tk是一个图形库，支持多个操作系统，使用Tcl语言开发；

Tk会调用操作系统提供的本地GUI接口，完成最终的GUI。

所以，我们的代码只需要调用Tkinter提供的接口就可以了。

### 第一个GUI程序

使用Tkinter十分简单，我们来编写一个GUI版本的“Hello, world!”。

第一步是导入Tkinter包的所有内容：

**from** tkinter **import** \*

第二步是从Frame派生一个Application类，这是所有Widget的父容器：

**class Application(Frame):**

**def** **\_\_init\_\_**(self, master=None):

Frame.\_\_init\_\_(self, master)

self.pack()

self.createWidgets()

**def** **createWidgets**(self):

self.helloLabel = Label(self, text='Hello, world!')

self.helloLabel.pack()

self.quitButton = Button(self, text='Quit', command=self.quit)

self.quitButton.pack()

在GUI中，每个Button、Label、输入框等，都是一个Widget。Frame则是可以容纳其他Widget的Widget，所有的Widget组合起来就是一棵树。

pack()方法把Widget加入到父容器中，并实现布局。pack()是最简单的布局，grid()可以实现更复杂的布局。

在createWidgets()方法中，我们创建一个Label和一个Button，当Button被点击时，触发self.quit()使程序退出。

第三步，实例化Application，并启动消息循环：

app = Application()

*# 设置窗口标题:*

app.master.title('Hello World')

*# 主消息循环:*

app.mainloop()

GUI程序的主线程负责监听来自操作系统的消息，并依次处理每一条消息。因此，如果消息处理非常耗时，就需要在新线程中处理。

运行这个GUI程序，可以看到下面的窗口：



点击“Quit”按钮或者窗口的“x”结束程序。

### 输入文本

我们再对这个GUI程序改进一下，加入一个文本框，让用户可以输入文本，然后点按钮后，弹出消息对话框。

**from** tkinter **import** \*

**import** tkinter.messagebox **as** messagebox

**class Application(Frame):**

**def** **\_\_init\_\_**(self, master=None):

Frame.\_\_init\_\_(self, master)

self.pack()

self.createWidgets()

**def** **createWidgets**(self):

self.nameInput = Entry(self)

self.nameInput.pack()

self.alertButton = Button(self, text='Hello', command=self.hello)

self.alertButton.pack()

**def** **hello**(self):

name = self.nameInput.get() **or** 'world'

messagebox.showinfo('Message', 'Hello, %s' % name)

app = Application()

*# 设置窗口标题:*

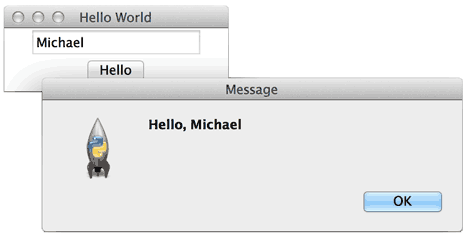
app.master.title('Hello World')

*# 主消息循环:*

app.mainloop()

当用户点击按钮时，触发hello()，通过self.nameInput.get()获得用户输入的文本后，使用tkMessageBox.showinfo()可以弹出消息对话框。

程序运行结果如下：



### 小结

Python内置的Tkinter可以满足基本的GUI程序的要求，如果是非常复杂的GUI程序，建议用操作系统原生支持的语言和库来编写。

### 参考源码

[hello\_gui.py](https://github.com/michaelliao/learn-python3/blob/master/samples/gui/hello_gui.py)

### 感觉本站内容不错，读后有收获？

#### 网络编程

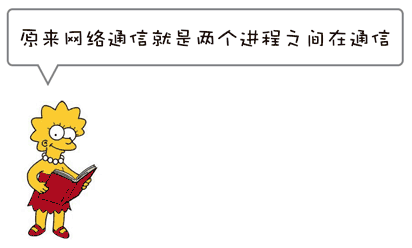
阅读: 34881

自从互联网诞生以来，现在基本上所有的程序都是网络程序，很少有单机版的程序了。

计算机网络就是把各个计算机连接到一起，让网络中的计算机可以互相通信。网络编程就是如何在程序中实现两台计算机的通信。

举个例子，当你使用浏览器访问新浪网时，你的计算机就和新浪的某台服务器通过互联网连接起来了，然后，新浪的服务器把网页内容作为数据通过互联网传输到你的电脑上。

由于你的电脑上可能不止浏览器，还有QQ、Skype、Dropbox、邮件客户端等，不同的程序连接的别的计算机也会不同，所以，更确切地说，网络通信是两台计算机上的两个进程之间的通信。比如，浏览器进程和新浪服务器上的某个Web服务进程在通信，而QQ进程是和腾讯的某个服务器上的某个进程在通信。



网络编程对所有开发语言都是一样的，Python也不例外。用Python进行网络编程，就是在Python程序本身这个进程内，连接别的服务器进程的通信端口进行通信。

本章我们将详细介绍Python网络编程的概念和最主要的两种网络类型的编程。

### 感觉本站内容不错，读后有收获？

#### TCP/IP简介

阅读: 56352

虽然大家现在对互联网很熟悉，但是计算机网络的出现比互联网要早很多。

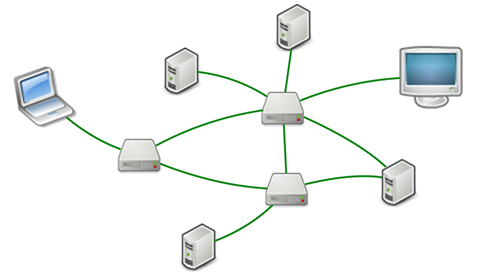
计算机为了联网，就必须规定通信协议，早期的计算机网络，都是由各厂商自己规定一套协议，IBM、Apple和Microsoft都有各自的网络协议，互不兼容，这就好比一群人有的说英语，有的说中文，有的说德语，说同一种语言的人可以交流，不同的语言之间就不行了。

为了把全世界的所有不同类型的计算机都连接起来，就必须规定一套全球通用的协议，为了实现互联网这个目标，互联网协议簇（Internet Protocol Suite）就是通用协议标准。Internet是由inter和net两个单词组合起来的，原意就是连接“网络”的网络，有了Internet，任何私有网络，只要支持这个协议，就可以联入互联网。

因为互联网协议包含了上百种协议标准，但是最重要的两个协议是TCP和IP协议，所以，大家把互联网的协议简称TCP/IP协议。

通信的时候，双方必须知道对方的标识，好比发邮件必须知道对方的邮件地址。互联网上每个计算机的唯一标识就是IP地址，类似123.123.123.123。如果一台计算机同时接入到两个或更多的网络，比如路由器，它就会有两个或多个IP地址，所以，IP地址对应的实际上是计算机的网络接口，通常是网卡。

IP协议负责把数据从一台计算机通过网络发送到另一台计算机。数据被分割成一小块一小块，然后通过IP包发送出去。由于互联网链路复杂，两台计算机之间经常有多条线路，因此，路由器就负责决定如何把一个IP包转发出去。IP包的特点是按块发送，途径多个路由，但不保证能到达，也不保证顺序到达。



IP地址实际上是一个32位整数（称为IPv4），以字符串表示的IP地址如192.168.0.1实际上是把32位整数按8位分组后的数字表示，目的是便于阅读。

IPv6地址实际上是一个128位整数，它是目前使用的IPv4的升级版，以字符串表示类似于2001:0db8:85a3:0042:1000:8a2e:0370:7334。

TCP协议则是建立在IP协议之上的。TCP协议负责在两台计算机之间建立可靠连接，保证数据包按顺序到达。TCP协议会通过握手建立连接，然后，对每个IP包编号，确保对方按顺序收到，如果包丢掉了，就自动重发。

许多常用的更高级的协议都是建立在TCP协议基础上的，比如用于浏览器的HTTP协议、发送邮件的SMTP协议等。

一个IP包除了包含要传输的数据外，还包含源IP地址和目标IP地址，源端口和目标端口。

端口有什么作用？在两台计算机通信时，只发IP地址是不够的，因为同一台计算机上跑着多个网络程序。一个IP包来了之后，到底是交给浏览器还是QQ，就需要端口号来区分。每个网络程序都向操作系统申请唯一的端口号，这样，两个进程在两台计算机之间建立网络连接就需要各自的IP地址和各自的端口号。

一个进程也可能同时与多个计算机建立链接，因此它会申请很多端口。

了解了TCP/IP协议的基本概念，IP地址和端口的概念，我们就可以开始进行网络编程了。

### 感觉本站内容不错，读后有收获？

#### TCP编程

阅读: 67587

Socket是网络编程的一个抽象概念。通常我们用一个Socket表示“打开了一个网络链接”，而打开一个Socket需要知道目标计算机的IP地址和端口号，再指定协议类型即可。

### 客户端

大多数连接都是可靠的TCP连接。创建TCP连接时，主动发起连接的叫客户端，被动响应连接的叫服务器。

举个例子，当我们在浏览器中访问新浪时，我们自己的计算机就是客户端，浏览器会主动向新浪的服务器发起连接。如果一切顺利，新浪的服务器接受了我们的连接，一个TCP连接就建立起来的，后面的通信就是发送网页内容了。

所以，我们要创建一个基于TCP连接的Socket，可以这样做：

*# 导入socket库:*

**import** socket

*# 创建一个socket:*

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

*# 建立连接:*

s.connect(('www.sina.com.cn', 80))

创建Socket时，AF\_INET指定使用IPv4协议，如果要用更先进的IPv6，就指定为AF\_INET6。SOCK\_STREAM指定使用面向流的TCP协议，这样，一个Socket对象就创建成功，但是还没有建立连接。

客户端要主动发起TCP连接，必须知道服务器的IP地址和端口号。新浪网站的IP地址可以用域名www.sina.com.cn自动转换到IP地址，但是怎么知道新浪服务器的端口号呢？

答案是作为服务器，提供什么样的服务，端口号就必须固定下来。由于我们想要访问网页，因此新浪提供网页服务的服务器必须把端口号固定在80端口，因为80端口是Web服务的标准端口。其他服务都有对应的标准端口号，例如SMTP服务是25端口，FTP服务是21端口，等等。端口号小于1024的是Internet标准服务的端口，端口号大于1024的，可以任意使用。

因此，我们连接新浪服务器的代码如下：

s.connect(('www.sina.com.cn', 80))

注意参数是一个tuple，包含地址和端口号。

建立TCP连接后，我们就可以向新浪服务器发送请求，要求返回首页的内容：

*# 发送数据:*

s.send(b'GET / HTTP/1.1\r\nHost: www.sina.com.cn\r\nConnection: close\r\n\r\n')

TCP连接创建的是双向通道，双方都可以同时给对方发数据。但是谁先发谁后发，怎么协调，要根据具体的协议来决定。例如，HTTP协议规定客户端必须先发请求给服务器，服务器收到后才发数据给客户端。

发送的文本格式必须符合HTTP标准，如果格式没问题，接下来就可以接收新浪服务器返回的数据了：

*# 接收数据:*

buffer = []

**while** **True**:

*# 每次最多接收1k字节:*

d = s.recv(1024)

**if** d:

buffer.append(d)

**else**:

**break**

data = b''.join(buffer)

接收数据时，调用recv(max)方法，一次最多接收指定的字节数，因此，在一个while循环中反复接收，直到recv()返回空数据，表示接收完毕，退出循环。

当我们接收完数据后，调用close()方法关闭Socket，这样，一次完整的网络通信就结束了：

*# 关闭连接:*

s.close()

接收到的数据包括HTTP头和网页本身，我们只需要把HTTP头和网页分离一下，把HTTP头打印出来，网页内容保存到文件：

header, html = data.split(b'\r\n\r\n', 1)

print(header.decode('utf-8'))

*# 把接收的数据写入文件:*

**with** open('sina.html', 'wb') **as** f:

f.write(html)

现在，只需要在浏览器中打开这个sina.html文件，就可以看到新浪的首页了。

### 服务器

和客户端编程相比，服务器编程就要复杂一些。

服务器进程首先要绑定一个端口并监听来自其他客户端的连接。如果某个客户端连接过来了，服务器就与该客户端建立Socket连接，随后的通信就靠这个Socket连接了。

所以，服务器会打开固定端口（比如80）监听，每来一个客户端连接，就创建该Socket连接。由于服务器会有大量来自客户端的连接，所以，服务器要能够区分一个Socket连接是和哪个客户端绑定的。一个Socket依赖4项：服务器地址、服务器端口、客户端地址、客户端端口来唯一确定一个Socket。

但是服务器还需要同时响应多个客户端的请求，所以，每个连接都需要一个新的进程或者新的线程来处理，否则，服务器一次就只能服务一个客户端了。

我们来编写一个简单的服务器程序，它接收客户端连接，把客户端发过来的字符串加上Hello再发回去。

首先，创建一个基于IPv4和TCP协议的Socket：

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

然后，我们要绑定监听的地址和端口。服务器可能有多块网卡，可以绑定到某一块网卡的IP地址上，也可以用0.0.0.0绑定到所有的网络地址，还可以用127.0.0.1绑定到本机地址。127.0.0.1是一个特殊的IP地址，表示本机地址，如果绑定到这个地址，客户端必须同时在本机运行才能连接，也就是说，外部的计算机无法连接进来。

端口号需要预先指定。因为我们写的这个服务不是标准服务，所以用9999这个端口号。请注意，小于1024的端口号必须要有管理员权限才能绑定：

*# 监听端口:*

s.bind(('127.0.0.1', 9999))

紧接着，调用listen()方法开始监听端口，传入的参数指定等待连接的最大数量：

s.listen(5)

**print**('Waiting for connection...')

接下来，服务器程序通过一个永久循环来接受来自客户端的连接，accept()会等待并返回一个客户端的连接:

**while** **True**:

*# 接受一个新连接:*

sock, addr = s.accept()

*# 创建新线程来处理TCP连接:*

t = threading.Thread(target=tcplink, args=(sock, addr))

t.start()

每个连接都必须创建新线程（或进程）来处理，否则，单线程在处理连接的过程中，无法接受其他客户端的连接：

**def** **tcplink**(sock, addr):

print('Accept new connection from %s:%s...' % addr)

sock.send(b'Welcome!')

**while** True:

data = sock.recv(1024)

time.sleep(1)

**if** **not** data **or** data.decode('utf-8') == 'exit':

**break**

sock.send(('Hello, %s!' % data.decode('utf-8')).encode('utf-8'))

sock.close()

print('Connection from %s:%s closed.' % addr)

连接建立后，服务器首先发一条欢迎消息，然后等待客户端数据，并加上Hello再发送给客户端。如果客户端发送了exit字符串，就直接关闭连接。

要测试这个服务器程序，我们还需要编写一个客户端程序：

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

*# 建立连接:*

s.connect(('127.0.0.1', 9999))

*# 接收欢迎消息:*

print(s.recv(1024).decode('utf-8'))

**for** data **in** [b'Michael', b'Tracy', b'Sarah']:

*# 发送数据:*

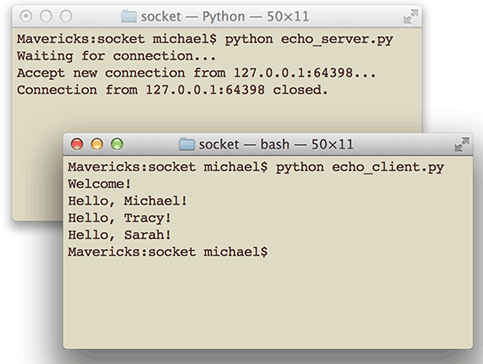
s.send(data)

print(s.recv(1024).decode('utf-8'))

s.send(b'exit')

s.close()

我们需要打开两个命令行窗口，一个运行服务器程序，另一个运行客户端程序，就可以看到效果了：



需要注意的是，客户端程序运行完毕就退出了，而服务器程序会永远运行下去，必须按Ctrl+C退出程序。

### 小结

用TCP协议进行Socket编程在Python中十分简单，对于客户端，要主动连接服务器的IP和指定端口，对于服务器，要首先监听指定端口，然后，对每一个新的连接，创建一个线程或进程来处理。通常，服务器程序会无限运行下去。

同一个端口，被一个Socket绑定了以后，就不能被别的Socket绑定了。

### 参考源码

[do\_tcp.py](https://github.com/michaelliao/learn-python3/blob/master/samples/socket/do_tcp.py)

### 感觉本站内容不错，读后有收获？

#### UDP编程

阅读: 15483

TCP是建立可靠连接，并且通信双方都可以以流的形式发送数据。相对TCP，UDP则是面向无连接的协议。

使用UDP协议时，不需要建立连接，只需要知道对方的IP地址和端口号，就可以直接发数据包。但是，能不能到达就不知道了。

虽然用UDP传输数据不可靠，但它的优点是和TCP比，速度快，对于不要求可靠到达的数据，就可以使用UDP协议。

我们来看看如何通过UDP协议传输数据。和TCP类似，使用UDP的通信双方也分为客户端和服务器。服务器首先需要绑定端口：

s = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

*# 绑定端口:*

s.bind(('127.0.0.1', 9999))

创建Socket时，SOCK\_DGRAM指定了这个Socket的类型是UDP。绑定端口和TCP一样，但是不需要调用listen()方法，而是直接接收来自任何客户端的数据：

**print**('Bind UDP on 9999...')

**while** **True**:

*# 接收数据:*

data, addr = s.recvfrom(1024)

**print**('Received from %s:%s.' % addr)

s.sendto(b'Hello, %s!' % data, addr)

recvfrom()方法返回数据和客户端的地址与端口，这样，服务器收到数据后，直接调用sendto()就可以把数据用UDP发给客户端。

注意这里省掉了多线程，因为这个例子很简单。

客户端使用UDP时，首先仍然创建基于UDP的Socket，然后，不需要调用connect()，直接通过sendto()给服务器发数据：

s = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

**for** data **in** [b'Michael', b'Tracy', b'Sarah']:

*# 发送数据:*

s.sendto(data, ('127.0.0.1', 9999))

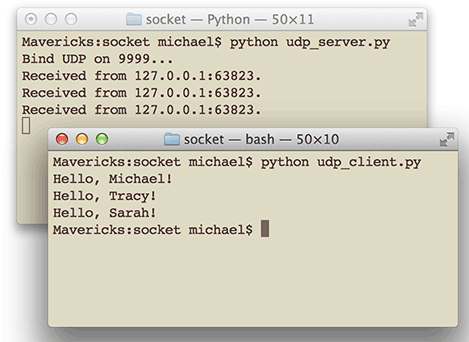
*# 接收数据:*

print(s.recv(1024).decode('utf-8'))

s.close()

从服务器接收数据仍然调用recv()方法。

仍然用两个命令行分别启动服务器和客户端测试，结果如下：



### 小结

UDP的使用与TCP类似，但是不需要建立连接。此外，服务器绑定UDP端口和TCP端口互不冲突，也就是说，UDP的9999端口与TCP的9999端口可以各自绑定。

### 参考源码

[udp\_server.py](https://github.com/michaelliao/learn-python3/blob/master/samples/socket/udp_server.py)

[udp\_client.py](https://github.com/michaelliao/learn-python3/blob/master/samples/socket/udp_client.py)

### 感觉本站内容不错，读后有收获？

#### 电子邮件

阅读: 5093

Email的历史比Web还要久远，直到现在，Email也是互联网上应用非常广泛的服务。

几乎所有的编程语言都支持发送和接收电子邮件，但是，先等等，在我们开始编写代码之前，有必要搞清楚电子邮件是如何在互联网上运作的。

我们来看看传统邮件是如何运作的。假设你现在在北京，要给一个香港的朋友发一封信，怎么做呢？

首先你得写好信，装进信封，写上地址，贴上邮票，然后就近找个邮局，把信仍进去。

信件会从就近的小邮局转运到大邮局，再从大邮局往别的城市发，比如先发到天津，再走海运到达香港，也可能走京九线到香港，但是你不用关心具体路线，你只需要知道一件事，就是信件走得很慢，至少要几天时间。

信件到达香港的某个邮局，也不会直接送到朋友的家里，因为邮局的叔叔是很聪明的，他怕你的朋友不在家，一趟一趟地白跑，所以，信件会投递到你的朋友的邮箱里，邮箱可能在公寓的一层，或者家门口，直到你的朋友回家的时候检查邮箱，发现信件后，就可以取到邮件了。

电子邮件的流程基本上也是按上面的方式运作的，只不过速度不是按天算，而是按秒算。

现在我们回到电子邮件，假设我们自己的电子邮件地址是me@163.com，对方的电子邮件地址是friend@sina.com（注意地址都是虚构的哈），现在我们用Outlook或者Foxmail之类的软件写好邮件，填上对方的Email地址，点“发送”，电子邮件就发出去了。这些电子邮件软件被称为**MUA**：Mail User Agent——邮件用户代理。

Email从MUA发出去，不是直接到达对方电脑，而是发到**MTA**：Mail Transfer Agent——邮件传输代理，就是那些Email服务提供商，比如网易、新浪等等。由于我们自己的电子邮件是163.com，所以，Email首先被投递到网易提供的MTA，再由网易的MTA发到对方服务商，也就是新浪的MTA。这个过程中间可能还会经过别的MTA，但是我们不关心具体路线，我们只关心速度。

Email到达新浪的MTA后，由于对方使用的是@sina.com的邮箱，因此，新浪的MTA会把Email投递到邮件的最终目的地**MDA**：Mail Delivery Agent——邮件投递代理。Email到达MDA后，就静静地躺在新浪的某个服务器上，存放在某个文件或特殊的数据库里，我们将这个长期保存邮件的地方称之为电子邮箱。

同普通邮件类似，Email不会直接到达对方的电脑，因为对方电脑不一定开机，开机也不一定联网。对方要取到邮件，必须通过MUA从MDA上把邮件取到自己的电脑上。

所以，一封电子邮件的旅程就是：

发件人 -> MUA -> MTA -> MTA -> 若干个MTA -> MDA <- MUA <- 收件人

有了上述基本概念，要编写程序来发送和接收邮件，本质上就是：

1. 编写MUA把邮件发到MTA；
2. 编写MUA从MDA上收邮件。

发邮件时，MUA和MTA使用的协议就是SMTP：Simple Mail Transfer Protocol，后面的MTA到另一个MTA也是用SMTP协议。

收邮件时，MUA和MDA使用的协议有两种：POP：Post Office Protocol，目前版本是3，俗称POP3；IMAP：Internet Message Access Protocol，目前版本是4，优点是不但能取邮件，还可以直接操作MDA上存储的邮件，比如从收件箱移到垃圾箱，等等。

邮件客户端软件在发邮件时，会让你先配置SMTP服务器，也就是你要发到哪个MTA上。假设你正在使用163的邮箱，你就不能直接发到新浪的MTA上，因为它只服务新浪的用户，所以，你得填163提供的SMTP服务器地址：smtp.163.com，为了证明你是163的用户，SMTP服务器还要求你填写邮箱地址和邮箱口令，这样，MUA才能正常地把Email通过SMTP协议发送到MTA。

类似的，从MDA收邮件时，MDA服务器也要求验证你的邮箱口令，确保不会有人冒充你收取你的邮件，所以，Outlook之类的邮件客户端会要求你填写POP3或IMAP服务器地址、邮箱地址和口令，这样，MUA才能顺利地通过POP或IMAP协议从MDA取到邮件。

在使用Python收发邮件前，请先准备好至少两个电子邮件，如xxx@163.com，xxx@sina.com，xxx@qq.com等，注意两个邮箱不要用同一家邮件服务商。

最后特别注意，目前大多数邮件服务商都需要手动打开SMTP发信和POP收信的功能，否则只允许在网页登录：



### 感觉本站内容不错，读后有收获？

#### SMTP发送邮件

阅读: 35522

SMTP是发送邮件的协议，Python内置对SMTP的支持，可以发送纯文本邮件、HTML邮件以及带附件的邮件。

Python对SMTP支持有smtplib和email两个模块，email负责构造邮件，smtplib负责发送邮件。

首先，我们来构造一个最简单的纯文本邮件：

**from** email.mime.text **import** MIMEText

msg = MIMEText('hello, send by Python...', 'plain', 'utf-8')

注意到构造MIMEText对象时，第一个参数就是邮件正文，第二个参数是MIME的subtype，传入'plain'表示纯文本，最终的MIME就是'text/plain'，最后一定要用utf-8编码保证多语言兼容性。

然后，通过SMTP发出去：

*# 输入Email地址和口令:*

from\_addr = input('From: ')

password = input('Password: ')

*# 输入收件人地址:*

to\_addr = input('To: ')

*# 输入SMTP服务器地址:*

smtp\_server = input('SMTP server: ')

**import** smtplib

server = smtplib.SMTP(smtp\_server, 25) *# SMTP协议默认端口是25*

server.set\_debuglevel(1)

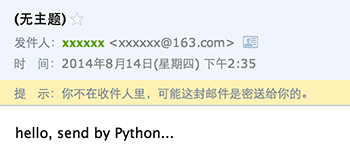
server.login(from\_addr, password)

server.sendmail(from\_addr, [to\_addr], msg.as\_string())

server.quit()

我们用set\_debuglevel(1)就可以打印出和SMTP服务器交互的所有信息。SMTP协议就是简单的文本命令和响应。login()方法用来登录SMTP服务器，sendmail()方法就是发邮件，由于可以一次发给多个人，所以传入一个list，邮件正文是一个str，as\_string()把MIMEText对象变成str。

如果一切顺利，就可以在收件人信箱中收到我们刚发送的Email：



仔细观察，发现如下问题：

1. 邮件没有主题；
2. 收件人的名字没有显示为友好的名字，比如Mr Green <green@example.com>；
3. 明明收到了邮件，却提示不在收件人中。

这是因为邮件主题、如何显示发件人、收件人等信息并不是通过SMTP协议发给MTA，而是包含在发给MTA的文本中的，所以，我们必须把From、To和Subject添加到MIMEText中，才是一封完整的邮件：

**from** email **import** encoders

**from** email.header **import** Header

**from** email.mime.text **import** MIMEText

**from** email.utils **import** parseaddr, formataddr

**import** smtplib

**def** **\_format\_addr**(s):

name, addr = parseaddr(s)

**return** formataddr((Header(name, 'utf-8').encode(), addr))

from\_addr = input('From: ')

password = input('Password: ')

to\_addr = input('To: ')

smtp\_server = input('SMTP server: ')

msg = MIMEText('hello, send by Python...', 'plain', 'utf-8')

msg['From'] = \_format\_addr('Python爱好者 <%s>' % from\_addr)

msg['To'] = \_format\_addr('管理员 <%s>' % to\_addr)

msg['Subject'] = Header('来自SMTP的问候……', 'utf-8').encode()

server = smtplib.SMTP(smtp\_server, 25)

server.set\_debuglevel(1)

server.login(from\_addr, password)

server.sendmail(from\_addr, [to\_addr], msg.as\_string())

server.quit()

我们编写了一个函数\_format\_addr()来格式化一个邮件地址。注意不能简单地传入name <addr@example.com>，因为如果包含中文，需要通过Header对象进行编码。

msg['To']接收的是字符串而不是list，如果有多个邮件地址，用,分隔即可。

再发送一遍邮件，就可以在收件人邮箱中看到正确的标题、发件人和收件人：



你看到的收件人的名字很可能不是我们传入的管理员，因为很多邮件服务商在显示邮件时，会把收件人名字自动替换为用户注册的名字，但是其他收件人名字的显示不受影响。

如果我们查看Email的原始内容，可以看到如下经过编码的邮件头：

From: =?utf-8?b?UHl0aG9u54ix5aW96ICF?= <xxxxxx@163.com>

To: =?utf-8?b?566h55CG5ZGY?= <xxxxxx@qq.com>

Subject: =?utf-8?b?5p2l6IeqU01UUOeahOmXruWAmeKApuKApg==?=

这就是经过Header对象编码的文本，包含utf-8编码信息和Base64编码的文本。如果我们自己来手动构造这样的编码文本，显然比较复杂。

### 发送HTML邮件

如果我们要发送HTML邮件，而不是普通的纯文本文件怎么办？方法很简单，在构造MIMEText对象时，把HTML字符串传进去，再把第二个参数由plain变为html就可以了：

msg = MIMEText('<html><body><h1>Hello</h1>' +

'<p>send by <a href="http://www.python.org">Python</a>...</p>' +

'</body></html>', 'html', 'utf-8')

再发送一遍邮件，你将看到以HTML显示的邮件：



### 发送附件

如果Email中要加上附件怎么办？带附件的邮件可以看做包含若干部分的邮件：文本和各个附件本身，所以，可以构造一个MIMEMultipart对象代表邮件本身，然后往里面加上一个MIMEText作为邮件正文，再继续往里面加上表示附件的MIMEBase对象即可：

*# 邮件对象:*

msg = MIMEMultipart()

msg['From'] = \_format\_addr('Python爱好者 <%s>' % from\_addr)

msg['To'] = \_format\_addr('管理员 <%s>' % to\_addr)

msg['Subject'] = Header('来自SMTP的问候……', 'utf-8').encode()

*# 邮件正文是MIMEText:*

msg.attach(MIMEText('send with file...', 'plain', 'utf-8'))

*# 添加附件就是加上一个MIMEBase，从本地读取一个图片:*

**with** open('/Users/michael/Downloads/test.png', 'rb') **as** f:

*# 设置附件的MIME和文件名，这里是png类型:*

mime = MIMEBase('image', 'png', filename='test.png')

*# 加上必要的头信息:*

mime.add\_header('Content-Disposition', 'attachment', filename='test.png')

mime.add\_header('Content-ID', '<0>')

mime.add\_header('X-Attachment-Id', '0')

*# 把附件的内容读进来:*

mime.set\_payload(f.read())

*# 用Base64编码:*

encoders.encode\_base64(mime)

*# 添加到MIMEMultipart:*

msg.attach(mime)

然后，按正常发送流程把msg（注意类型已变为MIMEMultipart）发送出去，就可以收到如下带附件的邮件：



### 发送图片

如果要把一个图片嵌入到邮件正文中怎么做？直接在HTML邮件中链接图片地址行不行？答案是，大部分邮件服务商都会自动屏蔽带有外链的图片，因为不知道这些链接是否指向恶意网站。

要把图片嵌入到邮件正文中，我们只需按照发送附件的方式，先把邮件作为附件添加进去，然后，在HTML中通过引用src="cid:0"就可以把附件作为图片嵌入了。如果有多个图片，给它们依次编号，然后引用不同的cid:x即可。

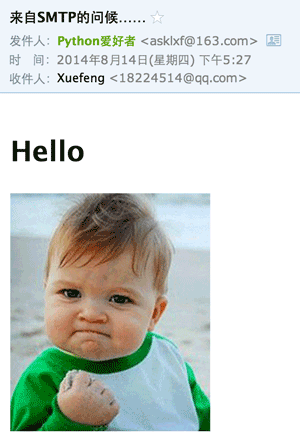
把上面代码加入MIMEMultipart的MIMEText从plain改为html，然后在适当的位置引用图片：

msg.attach(MIMEText('<html><body><h1>Hello</h1>' +

'<p><img src="cid:0"></p>' +

'</body></html>', 'html', 'utf-8'))

再次发送，就可以看到图片直接嵌入到邮件正文的效果：



### 同时支持HTML和Plain格式

如果我们发送HTML邮件，收件人通过浏览器或者Outlook之类的软件是可以正常浏览邮件内容的，但是，如果收件人使用的设备太古老，查看不了HTML邮件怎么办？

办法是在发送HTML的同时再附加一个纯文本，如果收件人无法查看HTML格式的邮件，就可以自动降级查看纯文本邮件。

利用MIMEMultipart就可以组合一个HTML和Plain，要注意指定subtype是alternative：

msg = MIMEMultipart('alternative')

msg['From'] = ...

msg['To'] = ...

msg['Subject'] = ...

msg.attach(MIMEText('hello', 'plain', 'utf-8'))

msg.attach(MIMEText('<html><body><h1>Hello</h1></body></html>', 'html', 'utf-8'))

# 正常发送msg对象...

### 加密SMTP

使用标准的25端口连接SMTP服务器时，使用的是明文传输，发送邮件的整个过程可能会被窃听。要更安全地发送邮件，可以加密SMTP会话，实际上就是先创建SSL安全连接，然后再使用SMTP协议发送邮件。

某些邮件服务商，例如Gmail，提供的SMTP服务必须要加密传输。我们来看看如何通过Gmail提供的安全SMTP发送邮件。

必须知道，Gmail的SMTP端口是587，因此，修改代码如下：

smtp\_server = 'smtp.gmail.com'

smtp\_port = 587

server = smtplib.SMTP(smtp\_server, smtp\_port)

server.starttls()

*# 剩下的代码和前面的一模一样:*

server.set\_debuglevel(1)

...

只需要在创建SMTP对象后，立刻调用starttls()方法，就创建了安全连接。后面的代码和前面的发送邮件代码完全一样。

如果因为网络问题无法连接Gmail的SMTP服务器，请相信我们的代码是没有问题的，你需要对你的网络设置做必要的调整。

### 小结

使用Python的smtplib发送邮件十分简单，只要掌握了各种邮件类型的构造方法，正确设置好邮件头，就可以顺利发出。

构造一个邮件对象就是一个Messag对象，如果构造一个MIMEText对象，就表示一个文本邮件对象，如果构造一个MIMEImage对象，就表示一个作为附件的图片，要把多个对象组合起来，就用MIMEMultipart对象，而MIMEBase可以表示任何对象。它们的继承关系如下：

Message

+- MIMEBase

+- MIMEMultipart

+- MIMENonMultipart

+- MIMEMessage

+- MIMEText

+- MIMEImage

这种嵌套关系就可以构造出任意复杂的邮件。你可以通过[email.mime文档](https://docs.python.org/3/library/email.mime.html)查看它们所在的包以及详细的用法。

### 参考源码

[send\_mail.py](https://github.com/michaelliao/learn-python3/blob/master/samples/mail/send_mail.py)

### 感觉本站内容不错，读后有收获？

#### POP3收取邮件

阅读: 1016

SMTP用于发送邮件，如果要收取邮件呢？

收取邮件就是编写一个**MUA**作为客户端，从**MDA**把邮件获取到用户的电脑或者手机上。收取邮件最常用的协议是**POP**协议，目前版本号是3，俗称**POP3**。

Python内置一个poplib模块，实现了POP3协议，可以直接用来收邮件。

注意到POP3协议收取的不是一个已经可以阅读的邮件本身，而是邮件的原始文本，这和SMTP协议很像，SMTP发送的也是经过编码后的一大段文本。

要把POP3收取的文本变成可以阅读的邮件，还需要用email模块提供的各种类来解析原始文本，变成可阅读的邮件对象。

所以，收取邮件分两步：

第一步：用poplib把邮件的原始文本下载到本地；

第二部：用email解析原始文本，还原为邮件对象。

### 通过POP3下载邮件

POP3协议本身很简单，以下面的代码为例，我们来获取最新的一封邮件内容：

import poplib

*# 输入邮件地址, 口令和POP3服务器地址:*

email = input('Email: ')

password = input('Password: ')

pop3\_server = input('POP3 server: ')

*# 连接到POP3服务器:*

server = poplib.POP3(pop3\_server)

*# 可以打开或关闭调试信息:*

server.set\_debuglevel(1)

*# 可选:打印POP3服务器的欢迎文字:*

**print**(server.getwelcome().decode('utf-8'))

*# 身份认证:*

server.user(email)

server.pass\_(password)

*# stat()返回邮件数量和占用空间:*

**print**('Messages: %s. Size: %s' % server.stat())

*# list()返回所有邮件的编号:*

resp, mails, octets = server.**list**()

*# 可以查看返回的列表类似[b'1 82923', b'2 2184', ...]*

**print**(mails)

*# 获取最新一封邮件, 注意索引号从1开始:*

index = len(mails)

resp, lines, octets = server.retr(index)

*# lines存储了邮件的原始文本的每一行,*

*# 可以获得整个邮件的原始文本:*

msg\_content = b'\r\n'.join(lines).decode('utf-8')

*# 稍后解析出邮件:*

msg = Parser().parsestr(msg\_content)

*# 可以根据邮件索引号直接从服务器删除邮件:*

*# server.dele(index)*

*# 关闭连接:*

server.quit()

用POP3获取邮件其实很简单，要获取所有邮件，只需要循环使用retr()把每一封邮件内容拿到即可。真正麻烦的是把邮件的原始内容解析为可以阅读的邮件对象。

### 解析邮件

解析邮件的过程和上一节构造邮件正好相反，因此，先导入必要的模块：

**from** email.parser **import** Parser

**from** email.header **import** decode\_header

**from** email.utils **import** parseaddr

**import** poplib

只需要一行代码就可以把邮件内容解析为Message对象：

msg = Parser().parsestr(msg\_content)

但是这个Message对象本身可能是一个MIMEMultipart对象，即包含嵌套的其他MIMEBase对象，嵌套可能还不止一层。

所以我们要递归地打印出Message对象的层次结构：

*# indent用于缩进显示:*

**def** **print\_info**(msg, indent=0):

**if** indent == 0:

**for** header **in** ['From', 'To', 'Subject']:

value = msg.get(header, '')

**if** value:

**if** header=='Subject':

value = decode\_str(value)

**else**:

hdr, addr = parseaddr(value)

name = decode\_str(hdr)

value = u'%s <%s>' % (name, addr)

print('%s%s: %s' % (' ' \* indent, header, value))

**if** (msg.is\_multipart()):

parts = msg.get\_payload()

**for** n, part **in** enumerate(parts):

print('%spart %s' % (' ' \* indent, n))

print('%s--------------------' % (' ' \* indent))

print\_info(part, indent + 1)

**else**:

content\_type = msg.get\_content\_type()

**if** content\_type=='text/plain' **or** content\_type=='text/html':

content = msg.get\_payload(decode=True)

charset = guess\_charset(msg)

**if** charset:

content = content.decode(charset)

print('%sText: %s' % (' ' \* indent, content + '...'))

**else**:

print('%sAttachment: %s' % (' ' \* indent, content\_type))

邮件的Subject或者Email中包含的名字都是经过编码后的str，要正常显示，就必须decode：

**def** **decode\_str**(s):

value, charset = decode\_header(s)[0]

**if** charset:

value = value.decode(charset)

**return** value

decode\_header()返回一个list，因为像Cc、Bcc这样的字段可能包含多个邮件地址，所以解析出来的会有多个元素。上面的代码我们偷了个懒，只取了第一个元素。

文本邮件的内容也是str，还需要检测编码，否则，非UTF-8编码的邮件都无法正常显示：

**def** **guess\_charset**(msg):

charset = msg.get\_charset()

**if** charset **is** None:

content\_type = msg.get('Content-Type', '').lower()

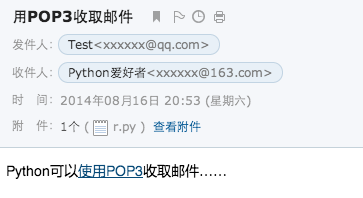
pos = content\_type.find('charset=')

**if** pos >= 0:

charset = content\_type[pos + 8:].strip()

**return** charset

把上面的代码整理好，我们就可以来试试收取一封邮件。先往自己的邮箱发一封邮件，然后用浏览器登录邮箱，看看邮件收到没，如果收到了，我们就来用Python程序把它收到本地：



运行程序，结果如下：

+OK Welcome to coremail Mail Pop3 Server (163coms[...])

Messages: 126. Size: 27228317

From: Test <xxxxxx@qq.com>

To: Python爱好者 <xxxxxx@163.com>

Subject: 用POP3收取邮件

part 0

--------------------

part 0

--------------------

Text: Python可以使用POP3收取邮件……...

part 1

--------------------

Text: Python可以<a href="...">使用POP3</a>收取邮件……...

part 1

--------------------

Attachment: application/octet-stream

我们从打印的结构可以看出，这封邮件是一个MIMEMultipart，它包含两部分：第一部分又是一个MIMEMultipart，第二部分是一个附件。而内嵌的MIMEMultipart是一个alternative类型，它包含一个纯文本格式的MIMEText和一个HTML格式的MIMEText。

### 小结

用Python的poplib模块收取邮件分两步：第一步是用POP3协议把邮件获取到本地，第二步是用email模块把原始邮件解析为Message对象，然后，用适当的形式把邮件内容展示给用户即可。

### 参考源码

[fetch\_mail.py](https://github.com/michaelliao/learn-python3/blob/master/samples/mail/fetch_mail.py)

### 感觉本站内容不错，读后有收获？

#### 访问数据库

阅读: 14359

程序运行的时候，数据都是在内存中的。当程序终止的时候，通常都需要将数据保存到磁盘上，无论是保存到本地磁盘，还是通过网络保存到服务器上，最终都会将数据写入磁盘文件。

而如何定义数据的存储格式就是一个大问题。如果我们自己来定义存储格式，比如保存一个班级所有学生的成绩单：

|  |  |
| --- | --- |
| **名字** | **成绩** |
| Michael | 99 |
| Bob | 85 |
| Bart | 59 |
| Lisa | 87 |

你可以用一个文本文件保存，一行保存一个学生，用,隔开：

Michael,99

Bob,85

Bart,59

Lisa,87

你还可以用JSON格式保存，也是文本文件：

[

{"name":"Michael","score":99},

{"name":"Bob","score":85},

{"name":"Bart","score":59},

{"name":"Lisa","score":87}

]

你还可以定义各种保存格式，但是问题来了：

存储和读取需要自己实现，JSON还是标准，自己定义的格式就各式各样了；

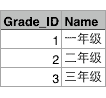
不能做快速查询，只有把数据全部读到内存中才能自己遍历，但有时候数据的大小远远超过了内存（比如蓝光电影，40GB的数据），根本无法全部读入内存。

为了便于程序保存和读取数据，而且，能直接通过条件快速查询到指定的数据，就出现了数据库（Database）这种专门用于集中存储和查询的软件。

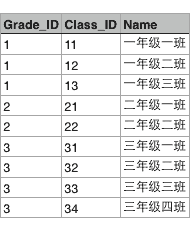
数据库软件诞生的历史非常久远，早在1950年数据库就诞生了。经历了网状数据库，层次数据库，我们现在广泛使用的关系数据库是20世纪70年代基于关系模型的基础上诞生的。

关系模型有一套复杂的数学理论，但是从概念上是十分容易理解的。举个学校的例子：

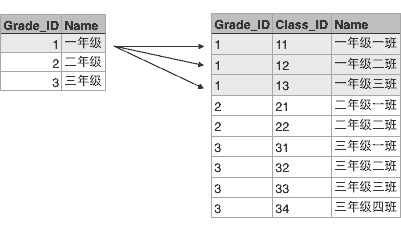
假设某个XX省YY市ZZ县第一实验小学有3个年级，要表示出这3个年级，可以在Excel中用一个表格画出来：



每个年级又有若干个班级，要把所有班级表示出来，可以在Excel中再画一个表格：



这两个表格有个映射关系，就是根据Grade\_ID可以在班级表中查找到对应的所有班级：



也就是Grade表的每一行对应Class表的多行，在关系数据库中，这种基于表（Table）的一对多的关系就是关系数据库的基础。

根据某个年级的ID就可以查找所有班级的行，这种查询语句在关系数据库中称为SQL语句，可以写成：

**SELECT** \* **FROM** classes **WHERE** grade\_id = '1';

结果也是一个表：

*---------+----------+----------*

grade\_id | class\_id | name

*---------+----------+----------*

1 | 11 | 一年级一班

*---------+----------+----------*

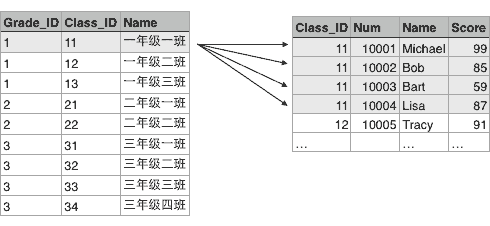
1 | 12 | 一年级二班

*---------+----------+----------*

1 | 13 | 一年级三班

*---------+----------+----------*

类似的，Class表的一行记录又可以关联到Student表的多行记录：



由于本教程不涉及到关系数据库的详细内容，如果你想从零学习关系数据库和基本的SQL语句，如果你想从零学习关系数据库和基本的SQL语句，请自行搜索相关课程。

### NoSQL

你也许还听说过NoSQL数据库，很多NoSQL宣传其速度和规模远远超过关系数据库，所以很多同学觉得有了NoSQL是否就不需要SQL了呢？千万不要被他们忽悠了，连SQL都不明白怎么可能搞明白NoSQL呢？

### 数据库类别

既然我们要使用关系数据库，就必须选择一个关系数据库。目前广泛使用的关系数据库也就这么几种：

付费的商用数据库：

* Oracle，典型的高富帅；
* SQL Server，微软自家产品，Windows定制专款；
* DB2，IBM的产品，听起来挺高端；
* Sybase，曾经跟微软是好基友，后来关系破裂，现在家境惨淡。

这些数据库都是不开源而且付费的，最大的好处是花了钱出了问题可以找厂家解决，不过在Web的世界里，常常需要部署成千上万的数据库服务器，当然不能把大把大把的银子扔给厂家，所以，无论是Google、Facebook，还是国内的BAT，无一例外都选择了免费的开源数据库：

* MySQL，大家都在用，一般错不了；
* PostgreSQL，学术气息有点重，其实挺不错，但知名度没有MySQL高；
* sqlite，嵌入式数据库，适合桌面和移动应用。

作为Python开发工程师，选择哪个免费数据库呢？当然是MySQL。因为MySQL普及率最高，出了错，可以很容易找到解决方法。而且，围绕MySQL有一大堆监控和运维的工具，安装和使用很方便。

为了能继续后面的学习，你需要从MySQL官方网站下载并安装[MySQL Community Server 5.6](http://dev.mysql.com/downloads/mysql/)，这个版本是免费的，其他高级版本是要收钱的（请放心，收钱的功能我们用不上）。

### 感觉本站内容不错，读后有收获？

#### 使用SQLite

阅读: 38710

SQLite是一种嵌入式数据库，它的数据库就是一个文件。由于SQLite本身是C写的，而且体积很小，所以，经常被集成到各种应用程序中，甚至在iOS和Android的App中都可以集成。

Python就内置了SQLite3，所以，在Python中使用SQLite，不需要安装任何东西，直接使用。

在使用SQLite前，我们先要搞清楚几个概念：

表是数据库中存放关系数据的集合，一个数据库里面通常都包含多个表，比如学生的表，班级的表，学校的表，等等。表和表之间通过外键关联。

要操作关系数据库，首先需要连接到数据库，一个数据库连接称为Connection；

连接到数据库后，需要打开游标，称之为Cursor，通过Cursor执行SQL语句，然后，获得执行结果。

Python定义了一套操作数据库的API接口，任何数据库要连接到Python，只需要提供符合Python标准的数据库驱动即可。

由于SQLite的驱动内置在Python标准库中，所以我们可以直接来操作SQLite数据库。

我们在Python交互式命令行实践一下：

*# 导入SQLite驱动:*

>>> **import** sqlite3

*# 连接到SQLite数据库*

*# 数据库文件是test.db*

*# 如果文件不存在，会自动在当前目录创建:*

>>> conn = sqlite3.connect('test.db')

*# 创建一个Cursor:*

>>> cursor = conn.cursor()

*# 执行一条SQL语句，创建user表:*

>>> cursor.execute('create table user (id varchar(20) primary key, name varchar(20))')

<sqlite3.Cursor object at 0x10f8aa260>

*# 继续执行一条SQL语句，插入一条记录:*

>>> cursor.execute('insert into user (id, name) values (\'1\', \'Michael\')')

<sqlite3.Cursor object at 0x10f8aa260>

*# 通过rowcount获得插入的行数:*

>>> cursor.rowcount

1

*# 关闭Cursor:*

>>> cursor.close()

*# 提交事务:*

>>> conn.commit()

*# 关闭Connection:*

>>> conn.close()

我们再试试查询记录：

>>> conn = sqlite3.connect('test.db')

>>> cursor = conn.cursor()

*# 执行查询语句:*

>>> cursor.execute('select \* from user where id=?', ('1',))

<sqlite3.Cursor object at 0x10f8aa340>

*# 获得查询结果集:*

>>> values = cursor.fetchall()

>>> values

[('1', 'Michael')]

>>> cursor.close()

>>> conn.close()

使用Python的DB-API时，只要搞清楚Connection和Cursor对象，打开后一定记得关闭，就可以放心地使用。

使用Cursor对象执行insert，update，delete语句时，执行结果由rowcount返回影响的行数，就可以拿到执行结果。

使用Cursor对象执行select语句时，通过featchall()可以拿到结果集。结果集是一个list，每个元素都是一个tuple，对应一行记录。

如果SQL语句带有参数，那么需要把参数按照位置传递给execute()方法，有几个?占位符就必须对应几个参数，例如：

cursor.execute('**select** \* **from** **user** **where** name=? **and** pwd=?', ('abc', 'password'))

SQLite支持常见的标准SQL语句以及几种常见的数据类型。具体文档请参阅SQLite官方网站。

### 小结

在Python中操作数据库时，要先导入数据库对应的驱动，然后，通过Connection对象和Cursor对象操作数据。

要确保打开的Connection对象和Cursor对象都正确地被关闭，否则，资源就会泄露。

如何才能确保出错的情况下也关闭掉Connection对象和Cursor对象呢？请回忆try:...except:...finally:...的用法。

### 练习

请编写函数，在Sqlite中根据分数段查找指定的名字：

窗体顶端

# -\*- coding: utf-8 -\*-

import os, sqlite3

db\_file = os.path.join(os.path.dirname(\_\_file\_\_), 'test.db')

if os.path.isfile(db\_file):

os.remove(db\_file)

# 初始数据:

conn = sqlite3.connect(db\_file)

cursor = conn.cursor()

cursor.execute('create table user(id varchar(20) primary key, name varchar(20), score int)')

cursor.execute(r"insert into user values ('A-001', 'Adam', 95)")

cursor.execute(r"insert into user values ('A-002', 'Bart', 62)")

cursor.execute(r"insert into user values ('A-003', 'Lisa', 78)")

cursor.close()

conn.commit()

conn.close()

def get\_score\_in(low, high):

' 返回指定分数区间的名字，按分数从低到高排序 '



# 测试:

assert get\_score\_in(80, 95) == ['Adam'], get\_score\_in(80, 95)

assert get\_score\_in(60, 80) == ['Bart', 'Lisa'], get\_score\_in(60, 80)

assert get\_score\_in(60, 100) == ['Bart', 'Lisa', 'Adam'], get\_score\_in(60, 100)

print('Pass')

窗体底端

### 参考源码

[do\_sqlite.py](https://github.com/michaelliao/learn-python3/blob/master/samples/db/do_sqlite.py)

### 感觉本站内容不错，读后有收获？

#### 使用MySQL

阅读: 69980

MySQL是Web世界中使用最广泛的数据库服务器。SQLite的特点是轻量级、可嵌入，但不能承受高并发访问，适合桌面和移动应用。而MySQL是为服务器端设计的数据库，能承受高并发访问，同时占用的内存也远远大于SQLite。

此外，MySQL内部有多种数据库引擎，最常用的引擎是支持数据库事务的InnoDB。

### 安装MySQL

可以直接从MySQL官方网站下载最新的[Community Server 5.6.x](http://dev.mysql.com/downloads/mysql/5.6.html)版本。MySQL是跨平台的，选择对应的平台下载安装文件，安装即可。

安装时，MySQL会提示输入root用户的口令，请务必记清楚。如果怕记不住，就把口令设置为password。

在Windows上，安装时请选择UTF-8编码，以便正确地处理中文。

在Mac或Linux上，需要编辑MySQL的配置文件，把数据库默认的编码全部改为UTF-8。MySQL的配置文件默认存放在/etc/my.cnf或者/etc/mysql/my.cnf：

**[client]**

default-character-set = utf8

**[mysqld]**

default-storage-engine = INNODB

character-set-server = utf8

collation-server = utf8\_general\_ci

重启MySQL后，可以通过MySQL的客户端命令行检查编码：

$ mysql -u root -p

Enter password:

Welcome to the MySQL monitor...

...

mysql> **show** variables **like** '%char%';

+*--------------------------+--------------------------------------------------------+*

| Variable\_name | Value |

+*--------------------------+--------------------------------------------------------+*

| character\_set\_client | utf8 |

| character\_set\_connection | utf8 |

| character\_set\_database | utf8 |

| character\_set\_filesystem | binary |

| character\_set\_results | utf8 |

| character\_set\_server | utf8 |

| character\_set\_system | utf8 |

| character\_sets\_dir | /usr/local/mysql-5.1.65-osx10.6-x86\_64/share/charsets/ |

+*--------------------------+--------------------------------------------------------+*

8 rows in **set** (0.00 sec)

看到utf8字样就表示编码设置正确。

注：如果MySQL的版本≥5.5.3，可以把编码设置为utf8mb4，utf8mb4和utf8完全兼容，但它支持最新的Unicode标准，可以显示emoji字符。

### 安装MySQL驱动

由于MySQL服务器以独立的进程运行，并通过网络对外服务，所以，需要支持Python的MySQL驱动来连接到MySQL服务器。MySQL官方提供了mysql-connector-python驱动，但是安装的时候需要给pip命令加上参数--allow-external：

$ pip install mysql-connector-python --allow-external mysql-connector-python

我们演示如何连接到MySQL服务器的test数据库：

*# 导入MySQL驱动:*

>>> **import** mysql.connector

*# 注意把password设为你的root口令:*

>>> conn = mysql.connector.connect(user='root', password='password', database='test')

>>> cursor = conn.cursor()

*# 创建user表:*

>>> cursor.execute('create table user (id varchar(20) primary key, name varchar(20))')

*# 插入一行记录，注意MySQL的占位符是%s:*

>>> cursor.execute('insert into user (id, name) values (%s, %s)', ['1', 'Michael'])

>>> cursor.rowcount

1

*# 提交事务:*

>>> conn.commit()

>>> cursor.close()

*# 运行查询:*

>>> cursor = conn.cursor()

>>> cursor.execute('select \* from user where id = %s', ('1',))

>>> values = cursor.fetchall()

>>> values

[('1', 'Michael')]

*# 关闭Cursor和Connection:*

>>> cursor.close()

True

>>> conn.close()

由于Python的DB-API定义都是通用的，所以，操作MySQL的数据库代码和SQLite类似。

### 小结

* 执行INSERT等操作后要调用commit()提交事务；
* MySQL的SQL占位符是%s。

### 参考源码

[do\_mysql.py](https://github.com/michaelliao/learn-python3/blob/master/samples/db/do_mysql.py)

### 感觉本站内容不错，读后有收获？

#### 使用SQLAlchemy

阅读: 19816

数据库表是一个二维表，包含多行多列。把一个表的内容用Python的数据结构表示出来的话，可以用一个list表示多行，list的每一个元素是tuple，表示一行记录，比如，包含id和name的user表：

[

('1', 'Michael'),

('2', 'Bob'),

('3', 'Adam')

]

Python的DB-API返回的数据结构就是像上面这样表示的。

但是用tuple表示一行很难看出表的结构。如果把一个tuple用class实例来表示，就可以更容易地看出表的结构来：

**class User(object):**

**def** **\_\_init\_\_**(self, id, name):

self.id = id

self.name = name

[

User('1', 'Michael'),

User('2', 'Bob'),

User('3', 'Adam')

]

这就是传说中的ORM技术：Object-Relational Mapping，把关系数据库的表结构映射到对象上。是不是很简单？

但是由谁来做这个转换呢？所以ORM框架应运而生。

在Python中，最有名的ORM框架是SQLAlchemy。我们来看看SQLAlchemy的用法。

首先通过pip安装SQLAlchemy：

$ pip install sqlalchemy

然后，利用上次我们在MySQL的test数据库中创建的user表，用SQLAlchemy来试试：

第一步，导入SQLAlchemy，并初始化DBSession：

*# 导入:*

**from** sqlalchemy **import** Column, String, create\_engine

**from** sqlalchemy.orm **import** sessionmaker

**from** sqlalchemy.ext.declarative **import** declarative\_base

*# 创建对象的基类:*

Base = declarative\_base()

*# 定义User对象:*

**class User(Base):**

*# 表的名字:*

\_\_tablename\_\_ = 'user'

*# 表的结构:*

id = Column(String(20), primary\_key=True)

name = Column(String(20))

*# 初始化数据库连接:*

engine = create\_engine('mysql+mysqlconnector://root:password@localhost:3306/test')

*# 创建DBSession类型:*

DBSession = sessionmaker(bind=engine)

以上代码完成SQLAlchemy的初始化和具体每个表的class定义。如果有多个表，就继续定义其他class，例如School：

**class School(Base):**

\_\_tablename\_\_ = 'school'

id = ...

name = ...

create\_engine()用来初始化数据库连接。SQLAlchemy用一个字符串表示连接信息：

'数据库类型+数据库驱动名称://用户名:口令@机器地址:端口号/数据库名'

你只需要根据需要替换掉用户名、口令等信息即可。

下面，我们看看如何向数据库表中添加一行记录。

由于有了ORM，我们向数据库表中添加一行记录，可以视为添加一个User对象：

*# 创建session对象:*

session = DBSession()

*# 创建新User对象:*

new\_user = User(id='5', name='Bob')

*# 添加到session:*

session.add(new\_user)

*# 提交即保存到数据库:*

session.commit()

*# 关闭session:*

session.close()

可见，关键是获取session，然后把对象添加到session，最后提交并关闭。DBSession对象可视为当前数据库连接。

如何从数据库表中查询数据呢？有了ORM，查询出来的可以不再是tuple，而是User对象。SQLAlchemy提供的查询接口如下：

*# 创建Session:*

session = DBSession()

*# 创建Query查询，filter是where条件，最后调用one()返回唯一行，如果调用all()则返回所有行:*

user = session.query(User).filter(User.id=='5').one()

*# 打印类型和对象的name属性:*

**print**('type:', type(user))

**print**('name:', user.name)

*# 关闭Session:*

session.close()

运行结果如下：

type: <class '\_\_main\_\_.User'>

name: Bob

可见，ORM就是把数据库表的行与相应的对象建立关联，互相转换。

由于关系数据库的多个表还可以用外键实现一对多、多对多等关联，相应地，ORM框架也可以提供两个对象之间的一对多、多对多等功能。

例如，如果一个User拥有多个Book，就可以定义一对多关系如下：

**class User(Base):**

\_\_tablename\_\_ = 'user'

id = Column(String(20), primary\_key=True)

name = Column(String(20))

*# 一对多:*

books = relationship('Book')

**class Book(Base):**

\_\_tablename\_\_ = 'book'

id = Column(String(20), primary\_key=True)

name = Column(String(20))

*# “多”的一方的book表是通过外键关联到user表的:*

user\_id = Column(String(20), ForeignKey('user.id'))

当我们查询一个User对象时，该对象的books属性将返回一个包含若干个Book对象的list。

### 小结

ORM框架的作用就是把数据库表的一行记录与一个对象互相做自动转换。

正确使用ORM的前提是了解关系数据库的原理。

### 参考源码

[do\_sqlalchemy.py](https://github.com/michaelliao/learn-python3/blob/master/samples/db/do_sqlalchemy.py)

### 感觉本站内容不错，读后有收获？

#### Web开发

阅读: 24485

最早的软件都是运行在大型机上的，软件使用者通过“哑终端”登陆到大型机上去运行软件。后来随着PC机的兴起，软件开始主要运行在桌面上，而数据库这样的软件运行在服务器端，这种Client/Server模式简称CS架构。

随着互联网的兴起，人们发现，CS架构不适合Web，最大的原因是Web应用程序的修改和升级非常迅速，而CS架构需要每个客户端逐个升级桌面App，因此，Browser/Server模式开始流行，简称BS架构。

在BS架构下，客户端只需要浏览器，应用程序的逻辑和数据都存储在服务器端。浏览器只需要请求服务器，获取Web页面，并把Web页面展示给用户即可。

当然，Web页面也具有极强的交互性。由于Web页面是用HTML编写的，而HTML具备超强的表现力，并且，服务器端升级后，客户端无需任何部署就可以使用到新的版本，因此，BS架构迅速流行起来。

今天，除了重量级的软件如Office，Photoshop等，大部分软件都以Web形式提供。比如，新浪提供的新闻、博客、微博等服务，均是Web应用。

Web应用开发可以说是目前软件开发中最重要的部分。Web开发也经历了好几个阶段：

1. 静态Web页面：由文本编辑器直接编辑并生成静态的HTML页面，如果要修改Web页面的内容，就需要再次编辑HTML源文件，早期的互联网Web页面就是静态的；
2. CGI：由于静态Web页面无法与用户交互，比如用户填写了一个注册表单，静态Web页面就无法处理。要处理用户发送的动态数据，出现了Common Gateway Interface，简称CGI，用C/C++编写。
3. ASP/JSP/PHP：由于Web应用特点是修改频繁，用C/C++这样的低级语言非常不适合Web开发，而脚本语言由于开发效率高，与HTML结合紧密，因此，迅速取代了CGI模式。ASP是微软推出的用VBScript脚本编程的Web开发技术，而JSP用Java来编写脚本，PHP本身则是开源的脚本语言。
4. MVC：为了解决直接用脚本语言嵌入HTML导致的可维护性差的问题，Web应用也引入了Model-View-Controller的模式，来简化Web开发。ASP发展为ASP.Net，JSP和PHP也有一大堆MVC框架。

目前，Web开发技术仍在快速发展中，异步开发、新的MVVM前端技术层出不穷。

Python的诞生历史比Web还要早，由于Python是一种解释型的脚本语言，开发效率高，所以非常适合用来做Web开发。

Python有上百种Web开发框架，有很多成熟的模板技术，选择Python开发Web应用，不但开发效率高，而且运行速度快。

本章我们会详细讨论Python Web开发技术。

### 感觉本站内容不错，读后有收获？

#### HTTP协议简介

阅读: 57235

在Web应用中，服务器把网页传给浏览器，实际上就是把网页的HTML代码发送给浏览器，让浏览器显示出来。而浏览器和服务器之间的传输协议是HTTP，所以：

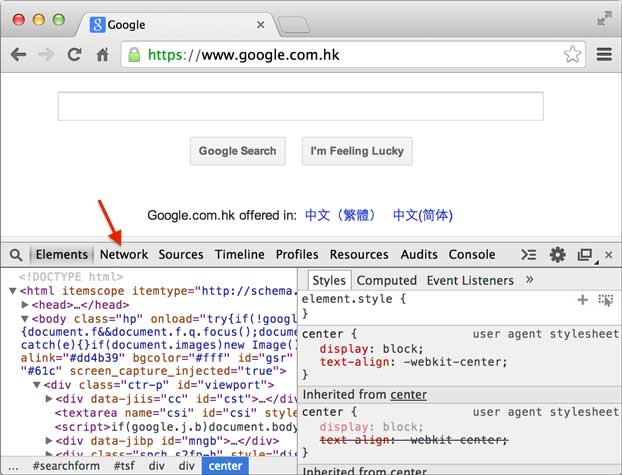
* HTML是一种用来定义网页的文本，会HTML，就可以编写网页；
* HTTP是在网络上传输HTML的协议，用于浏览器和服务器的通信。

在举例子之前，我们需要安装Google的[Chrome浏览器](http://www.google.com/intl/zh-CN/chrome/)。

为什么要使用Chrome浏览器而不是IE呢？因为IE实在是太慢了，并且，IE对于开发和调试Web应用程序完全是一点用也没有。

我们需要在浏览器很方便地调试我们的Web应用，而Chrome提供了一套完整地调试工具，非常适合Web开发。

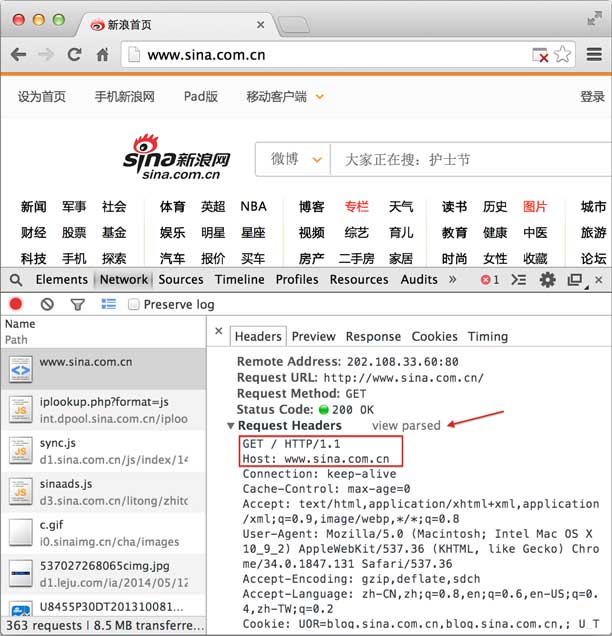
安装好Chrome浏览器后，打开Chrome，在菜单中选择“视图”，“开发者”，“开发者工具”，就可以显示开发者工具：



Elements显示网页的结构，Network显示浏览器和服务器的通信。我们点Network，确保第一个小红灯亮着，Chrome就会记录所有浏览器和服务器之间的通信：



当我们在地址栏输入www.sina.com.cn时，浏览器将显示新浪的首页。在这个过程中，浏览器都干了哪些事情呢？通过Network的记录，我们就可以知道。在Network中，定位到第一条记录，点击，右侧将显示Request Headers，点击右侧的view source，我们就可以看到浏览器发给新浪服务器的请求：



最主要的头两行分析如下，第一行：

GET / HTTP/1.1

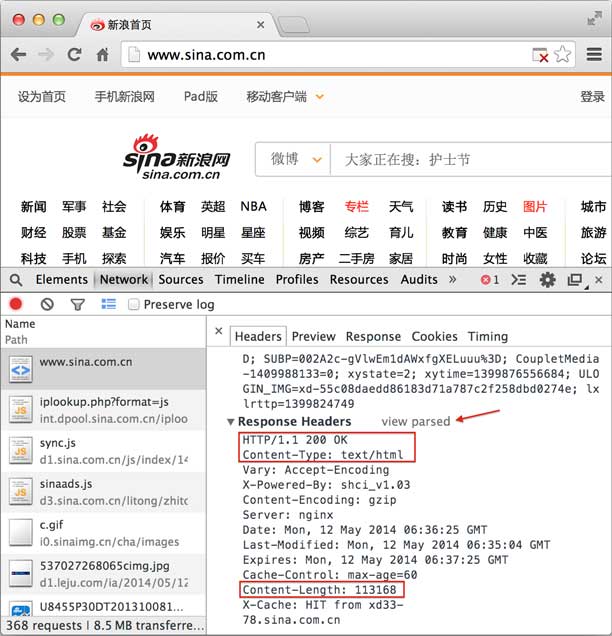
GET表示一个读取请求，将从服务器获得网页数据，/表示URL的路径，URL总是以/开头，/就表示首页，最后的HTTP/1.1指示采用的HTTP协议版本是1.1。目前HTTP协议的版本就是1.1，但是大部分服务器也支持1.0版本，主要区别在于1.1版本允许多个HTTP请求复用一个TCP连接，以加快传输速度。

从第二行开始，每一行都类似于Xxx: abcdefg：

Host: www.sina.com.cn

表示请求的域名是www.sina.com.cn。如果一台服务器有多个网站，服务器就需要通过Host来区分浏览器请求的是哪个网站。

继续往下找到Response Headers，点击view source，显示服务器返回的原始响应数据：



HTTP响应分为Header和Body两部分（Body是可选项），我们在Network中看到的Header最重要的几行如下：

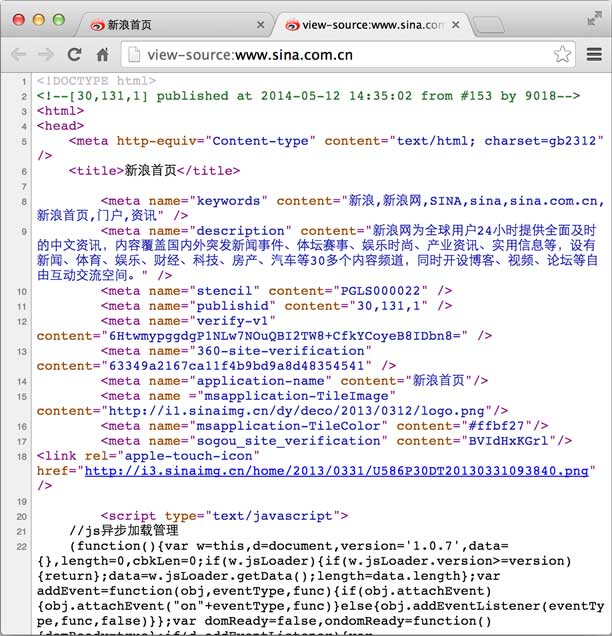
200 OK

200表示一个成功的响应，后面的OK是说明。失败的响应有404 Not Found：网页不存在，500 Internal Server Error：服务器内部出错，等等。

Content-Type: text/html

Content-Type指示响应的内容，这里是text/html表示HTML网页。请注意，浏览器就是依靠Content-Type来判断响应的内容是网页还是图片，是视频还是音乐。浏览器并不靠URL来判断响应的内容，所以，即使URL是http://example.com/abc.jpg，它也不一定就是图片。

HTTP响应的Body就是HTML源码，我们在菜单栏选择“视图”，“开发者”，“查看网页源码”就可以在浏览器中直接查看HTML源码：



当浏览器读取到新浪首页的HTML源码后，它会解析HTML，显示页面，然后，根据HTML里面的各种链接，再发送HTTP请求给新浪服务器，拿到相应的图片、视频、Flash、JavaScript脚本、CSS等各种资源，最终显示出一个完整的页面。所以我们在Network下面能看到很多额外的HTTP请求。

### HTTP请求

跟踪了新浪的首页，我们来总结一下HTTP请求的流程：

步骤1：浏览器首先向服务器发送HTTP请求，请求包括：

方法：GET还是POST，GET仅请求资源，POST会附带用户数据；

路径：/full/url/path；

域名：由Host头指定：Host: www.sina.com.cn

以及其他相关的Header；

如果是POST，那么请求还包括一个Body，包含用户数据。

步骤2：服务器向浏览器返回HTTP响应，响应包括：

响应代码：200表示成功，3xx表示重定向，4xx表示客户端发送的请求有错误，5xx表示服务器端处理时发生了错误；

响应类型：由Content-Type指定；

以及其他相关的Header；

通常服务器的HTTP响应会携带内容，也就是有一个Body，包含响应的内容，网页的HTML源码就在Body中。

步骤3：如果浏览器还需要继续向服务器请求其他资源，比如图片，就再次发出HTTP请求，重复步骤1、2。

Web采用的HTTP协议采用了非常简单的请求-响应模式，从而大大简化了开发。当我们编写一个页面时，我们只需要在HTTP请求中把HTML发送出去，不需要考虑如何附带图片、视频等，浏览器如果需要请求图片和视频，它会发送另一个HTTP请求，因此，一个HTTP请求只处理一个资源。

HTTP协议同时具备极强的扩展性，虽然浏览器请求的是http://www.sina.com.cn/的首页，但是新浪在HTML中可以链入其他服务器的资源，比如<img src="http://i1.sinaimg.cn/home/2013/1008/U8455P30DT20131008135420.png">，从而将请求压力分散到各个服务器上，并且，一个站点可以链接到其他站点，无数个站点互相链接起来，就形成了World Wide Web，简称WWW。

### HTTP格式

每个HTTP请求和响应都遵循相同的格式，一个HTTP包含Header和Body两部分，其中Body是可选的。

HTTP协议是一种文本协议，所以，它的格式也非常简单。HTTP GET请求的格式：

GET /path HTTP/1.1

Header1: Value1

Header2: Value2

Header3: Value3

每个Header一行一个，换行符是\r\n。

HTTP POST请求的格式：

POST /path HTTP/1.1

Header1: Value1

Header2: Value2

Header3: Value3

body data goes here...

当遇到连续两个\r\n时，Header部分结束，后面的数据全部是Body。

HTTP响应的格式：

200 OK

Header1: Value1

Header2: Value2

Header3: Value3

body data goes here...

HTTP响应如果包含body，也是通过\r\n\r\n来分隔的。请再次注意，Body的数据类型由Content-Type头来确定，如果是网页，Body就是文本，如果是图片，Body就是图片的二进制数据。

当存在Content-Encoding时，Body数据是被压缩的，最常见的压缩方式是gzip，所以，看到Content-Encoding: gzip时，需要将Body数据先解压缩，才能得到真正的数据。压缩的目的在于减少Body的大小，加快网络传输。

要详细了解HTTP协议，推荐“[HTTP: The Definitive Guide](http://shop.oreilly.com/product/9781565925090.do)”一书，非常不错，有中文译本：

[HTTP权威指南](http://t.cn/R7FguRq)

### 感觉本站内容不错，读后有收获？

#### HTML简介

阅读: 21481

网页就是HTML？这么理解大概没错。因为网页中不但包含文字，还有图片、视频、Flash小游戏，有复杂的排版、动画效果，所以，HTML定义了一套语法规则，来告诉浏览器如何把一个丰富多彩的页面显示出来。

HTML长什么样？上次我们看了新浪首页的HTML源码，如果仔细数数，竟然有6000多行！

所以，学HTML，就不要指望从新浪入手了。我们来看看最简单的HTML长什么样：

<html>

<head>

<title>Hello</title>

</head>

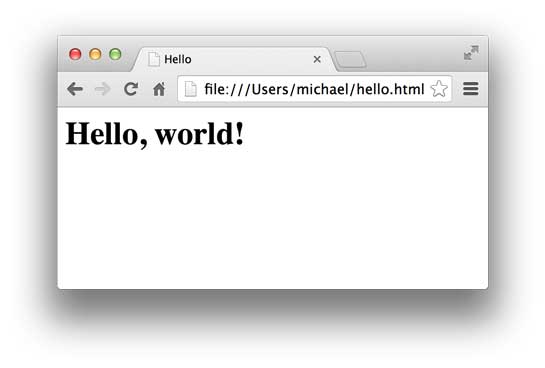
<body>

<h1>Hello, world!</h1>

</body>

</html>

可以用文本编辑器编写HTML，然后保存为hello.html，双击或者把文件拖到浏览器中，就可以看到效果：



HTML文档就是一系列的Tag组成，最外层的Tag是<html>。规范的HTML也包含<head>...</head>和<body>...</body>（注意不要和HTTP的Header、Body搞混了），由于HTML是富文档模型，所以，还有一系列的Tag用来表示链接、图片、表格、表单等等。

### CSS简介

CSS是Cascading Style Sheets（层叠样式表）的简称，CSS用来控制HTML里的所有元素如何展现，比如，给标题元素<h1>加一个样式，变成48号字体，灰色，带阴影：

<html>

<head>

<title>Hello</title>

<style>

h1 {

color: #333333;

font-size: 48px;

text-shadow: 3px 3px 3px #666666;

}

</style>

</head>

<body>

<h1>Hello, world!</h1>

</body>

</html>

效果如下：



### JavaScript简介

JavaScript虽然名称有个Java，但它和Java真的一点关系没有。JavaScript是为了让HTML具有交互性而作为脚本语言添加的，JavaScript既可以内嵌到HTML中，也可以从外部链接到HTML中。如果我们希望当用户点击标题时把标题变成红色，就必须通过JavaScript来实现：

<html>

<head>

<title>Hello</title>

<style>

h1 {

color: #333333;

font-size: 48px;

text-shadow: 3px 3px 3px #666666;

}

</style>

<script>

**function** change() {

document.getElementsByTagName('h1')[0].style.color = '#ff0000';

}

</script>

</head>

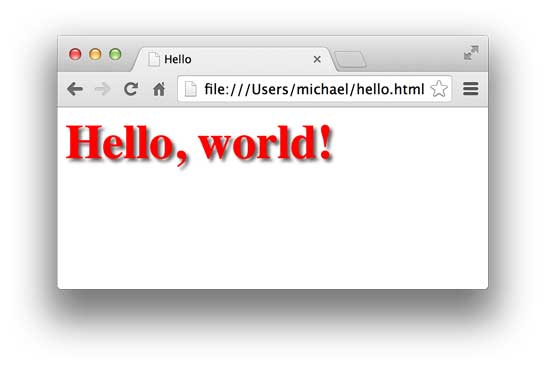
<body>

<h1 onclick="change()">Hello, world!</h1>

</body>

</html>

点击标题后效果如下：



### 小结

如果要学习Web开发，首先要对HTML、CSS和JavaScript作一定的了解。HTML定义了页面的内容，CSS来控制页面元素的样式，而JavaScript负责页面的交互逻辑。

讲解HTML、CSS和JavaScript就可以写3本书，对于优秀的Web开发人员来说，精通HTML、CSS和JavaScript是必须的，这里推荐一个在线学习网站w3schools：

<http://www.w3schools.com/>

以及一个对应的中文版本：

<http://www.w3school.com.cn/>

当我们用Python或者其他语言开发Web应用时，我们就是要在服务器端动态创建出HTML，这样，浏览器就会向不同的用户显示出不同的Web页面。

### 感觉本站内容不错，读后有收获？

#### WSGI接口

阅读: 42991

了解了HTTP协议和HTML文档，我们其实就明白了一个Web应用的本质就是：

1. 浏览器发送一个HTTP请求；
2. 服务器收到请求，生成一个HTML文档；
3. 服务器把HTML文档作为HTTP响应的Body发送给浏览器；
4. 浏览器收到HTTP响应，从HTTP Body取出HTML文档并显示。

所以，最简单的Web应用就是先把HTML用文件保存好，用一个现成的HTTP服务器软件，接收用户请求，从文件中读取HTML，返回。Apache、Nginx、Lighttpd等这些常见的静态服务器就是干这件事情的。

如果要动态生成HTML，就需要把上述步骤自己来实现。不过，接受HTTP请求、解析HTTP请求、发送HTTP响应都是苦力活，如果我们自己来写这些底层代码，还没开始写动态HTML呢，就得花个把月去读HTTP规范。

正确的做法是底层代码由专门的服务器软件实现，我们用Python专注于生成HTML文档。因为我们不希望接触到TCP连接、HTTP原始请求和响应格式，所以，需要一个统一的接口，让我们专心用Python编写Web业务。

这个接口就是WSGI：Web Server Gateway Interface。

WSGI接口定义非常简单，它只要求Web开发者实现一个函数，就可以响应HTTP请求。我们来看一个最简单的Web版本的“Hello, web!”：

**def** **application**(environ, start\_response):

start\_response('200 OK', [('Content-Type', 'text/html')])

**return** [b'<h1>Hello, web!</h1>']

上面的application()函数就是符合WSGI标准的一个HTTP处理函数，它接收两个参数：

* environ：一个包含所有HTTP请求信息的dict对象；
* start\_response：一个发送HTTP响应的函数。

在application()函数中，调用：

start\_response('200 OK', [('Content-Type', 'text/html')])

就发送了HTTP响应的Header，注意Header只能发送一次，也就是只能调用一次start\_response()函数。start\_response()函数接收两个参数，一个是HTTP响应码，一个是一组list表示的HTTP Header，每个Header用一个包含两个str的tuple表示。

通常情况下，都应该把Content-Type头发送给浏览器。其他很多常用的HTTP Header也应该发送。

然后，函数的返回值b'<h1>Hello, web!</h1>'将作为HTTP响应的Body发送给浏览器。

有了WSGI，我们关心的就是如何从environ这个dict对象拿到HTTP请求信息，然后构造HTML，通过start\_response()发送Header，最后返回Body。

整个application()函数本身没有涉及到任何解析HTTP的部分，也就是说，底层代码不需要我们自己编写，我们只负责在更高层次上考虑如何响应请求就可以了。

不过，等等，这个application()函数怎么调用？如果我们自己调用，两个参数environ和start\_response我们没法提供，返回的bytes也没法发给浏览器。

所以application()函数必须由WSGI服务器来调用。有很多符合WSGI规范的服务器，我们可以挑选一个来用。但是现在，我们只想尽快测试一下我们编写的application()函数真的可以把HTML输出到浏览器，所以，要赶紧找一个最简单的WSGI服务器，把我们的Web应用程序跑起来。

好消息是Python内置了一个WSGI服务器，这个模块叫wsgiref，它是用纯Python编写的WSGI服务器的参考实现。所谓“参考实现”是指该实现完全符合WSGI标准，但是不考虑任何运行效率，仅供开发和测试使用。

### 运行WSGI服务

我们先编写hello.py，实现Web应用程序的WSGI处理函数：

*# hello.py*

**def** **application**(environ, start\_response):

start\_response('200 OK', [('Content-Type', 'text/html')])

**return** [b'<h1>Hello, web!</h1>']

然后，再编写一个server.py，负责启动WSGI服务器，加载application()函数：

*# server.py*

*# 从wsgiref模块导入:*

**from** wsgiref.simple\_server **import** make\_server

*# 导入我们自己编写的application函数:*

**from** hello **import** application

*# 创建一个服务器，IP地址为空，端口是8000，处理函数是application:*

httpd = make\_server('', 8000, application)

print('Serving HTTP on port 8000...')

*# 开始监听HTTP请求:*

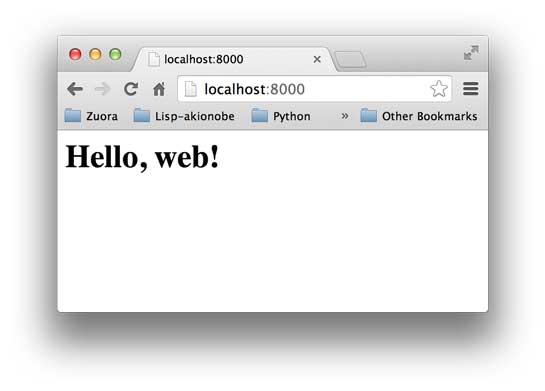
httpd.serve\_forever()

确保以上两个文件在同一个目录下，然后在命令行输入python server.py来启动WSGI服务器：

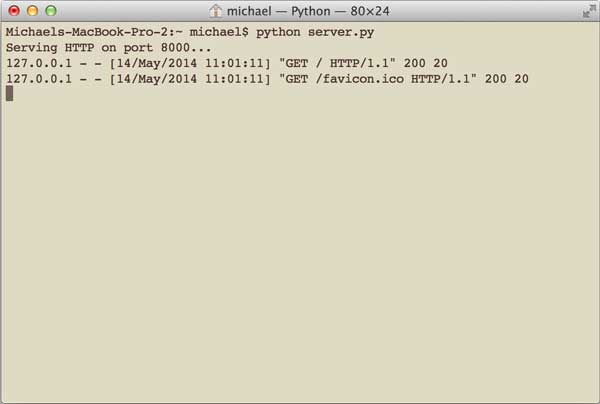


注意：如果8000端口已被其他程序占用，启动将失败，请修改成其他端口。

启动成功后，打开浏览器，输入http://localhost:8000/，就可以看到结果了：



在命令行可以看到wsgiref打印的log信息：



按Ctrl+C终止服务器。

如果你觉得这个Web应用太简单了，可以稍微改造一下，从environ里读取PATH\_INFO，这样可以显示更加动态的内容：

*# hello.py*

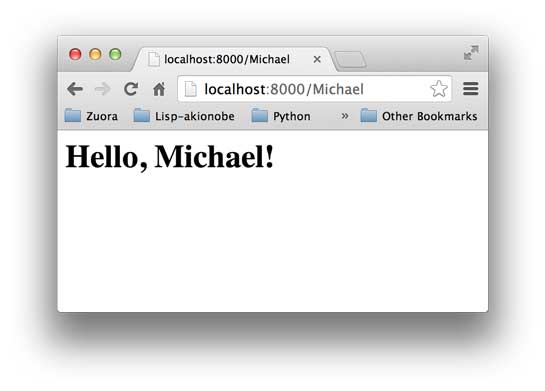
**def** **application**(environ, start\_response):

start\_response('200 OK', [('Content-Type', 'text/html')])

body = '<h1>Hello, %s!</h1>' % (environ['PATH\_INFO'][1:] **or** 'web')

**return** [body.encode('utf-8')]

你可以在地址栏输入用户名作为URL的一部分，将返回Hello, xxx!：



是不是有点Web App的感觉了？

### 小结

无论多么复杂的Web应用程序，入口都是一个WSGI处理函数。HTTP请求的所有输入信息都可以通过environ获得，HTTP响应的输出都可以通过start\_response()加上函数返回值作为Body。

复杂的Web应用程序，光靠一个WSGI函数来处理还是太底层了，我们需要在WSGI之上再抽象出Web框架，进一步简化Web开发。

### 参考源码

[hello.py](https://github.com/michaelliao/learn-python3/blob/master/samples/web/hello.py)

[do\_wsgi.py](https://github.com/michaelliao/learn-python3/blob/master/samples/web/do_wsgi.py)

### 感觉本站内容不错，读后有收获？

#### 使用Web框架

阅读: 55725

了解了WSGI框架，我们发现：其实一个Web App，就是写一个WSGI的处理函数，针对每个HTTP请求进行响应。

但是如何处理HTTP请求不是问题，问题是如何处理100个不同的URL。

每一个URL可以对应GET和POST请求，当然还有PUT、DELETE等请求，但是我们通常只考虑最常见的GET和POST请求。

一个最简单的想法是从environ变量里取出HTTP请求的信息，然后逐个判断：

**def** **application**(environ, start\_response):

method = environ['REQUEST\_METHOD']

path = environ['PATH\_INFO']

**if** method=='GET' **and** path=='/':

**return** handle\_home(environ, start\_response)

**if** method=='POST' **and** path='/signin':

**return** handle\_signin(environ, start\_response)

...

只是这么写下去代码是肯定没法维护了。

代码这么写没法维护的原因是因为WSGI提供的接口虽然比HTTP接口高级了不少，但和Web App的处理逻辑比，还是比较低级，我们需要在WSGI接口之上能进一步抽象，让我们专注于用一个函数处理一个URL，至于URL到函数的映射，就交给Web框架来做。

由于用Python开发一个Web框架十分容易，所以Python有上百个开源的Web框架。这里我们先不讨论各种Web框架的优缺点，直接选择一个比较流行的Web框架——[Flask](http://flask.pocoo.org/)来使用。

用Flask编写Web App比WSGI接口简单（这不是废话么，要是比WSGI还复杂，用框架干嘛？），我们先用pip安装Flask：

$ pip install flask

然后写一个app.py，处理3个URL，分别是：

* GET /：首页，返回Home；
* GET /signin：登录页，显示登录表单；
* POST /signin：处理登录表单，显示登录结果。

注意噢，同一个URL/signin分别有GET和POST两种请求，映射到两个处理函数中。

Flask通过Python的[装饰器](http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/0014318435599930270c0381a3b44db991cd6d858064ac0000)在内部自动地把URL和函数给关联起来，所以，我们写出来的代码就像这样：

from flask import Flask

from flask import request

app = Flask(\_\_name\_\_)

@app.route('/', methods=['GET', 'POST'])

def home():

return '<h1>Home</h1>'

@app.route('/signin', methods=['GET'])

def signin\_form():

return '''<form action="/signin" method="post">

<p><input name="username"></p>

<p><input name="password" type="password"></p>

<p><button type="submit">Sign In</button></p>

</form>'''

@app.route('/signin', methods=['POST'])

def signin():

# 需要从request对象读取表单内容：

if request.form['username']=='admin' and request.form['password']=='password':

return '<h3>Hello, admin!</h3>'

return '<h3>Bad username or password.</h3>'

if \_\_name\_\_ == '\_\_main\_\_':

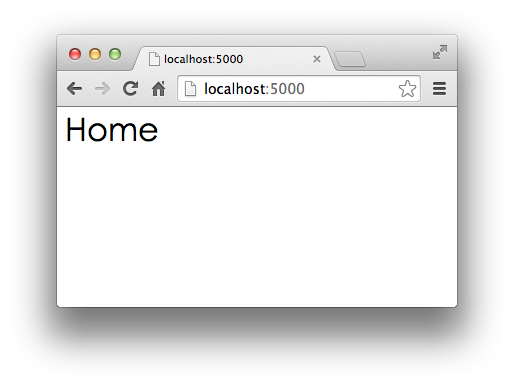
app.run()

运行python app.py，Flask自带的Server在端口5000上监听：

$ python app.py

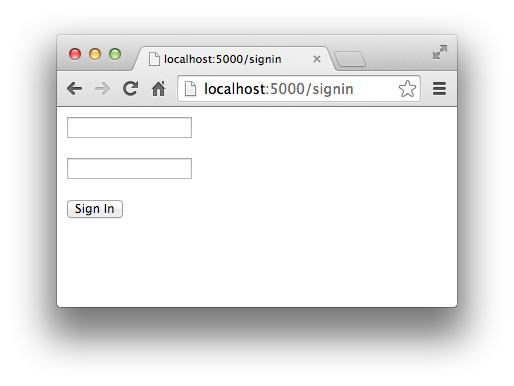
\* Running on http://127.0.0.1:5000/

打开浏览器，输入首页地址http://localhost:5000/：

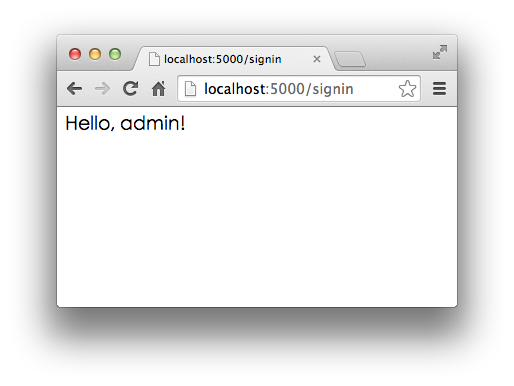


首页显示正确！

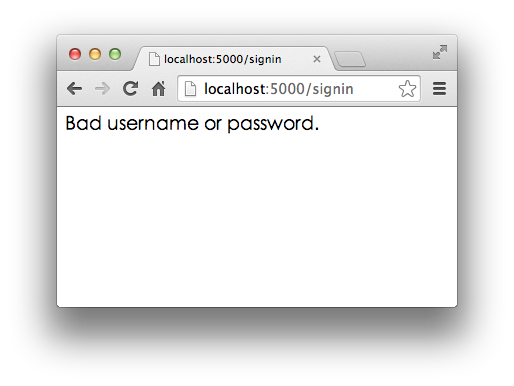
再在浏览器地址栏输入http://localhost:5000/signin，会显示登录表单：



输入预设的用户名admin和口令password，登录成功：



输入其他错误的用户名和口令，登录失败：



实际的Web App应该拿到用户名和口令后，去数据库查询再比对，来判断用户是否能登录成功。

除了Flask，常见的Python Web框架还有：

* [Django](https://www.djangoproject.com/)：全能型Web框架；
* [web.py](http://webpy.org/)：一个小巧的Web框架；
* [Bottle](http://bottlepy.org/)：和Flask类似的Web框架；
* [Tornado](http://www.tornadoweb.org/)：Facebook的开源异步Web框架。

当然了，因为开发Python的Web框架也不是什么难事，我们后面也会讲到开发Web框架的内容。

### 小结

有了Web框架，我们在编写Web应用时，注意力就从WSGI处理函数转移到URL+对应的处理函数，这样，编写Web App就更加简单了。

在编写URL处理函数时，除了配置URL外，从HTTP请求拿到用户数据也是非常重要的。Web框架都提供了自己的API来实现这些功能。Flask通过request.form['name']来获取表单的内容。

### 参考源码

[do\_flask.py](https://github.com/michaelliao/learn-python3/blob/master/samples/web/do_flask.py)

### 感觉本站内容不错，读后有收获？

#### 使用模板

阅读: 26413

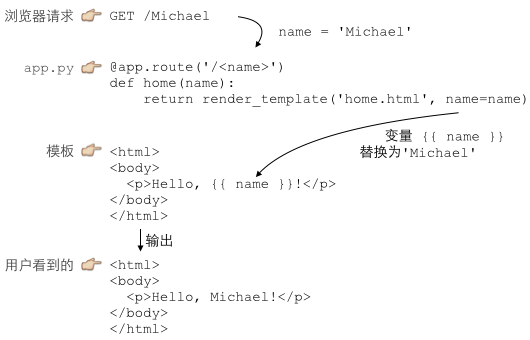
Web框架把我们从WSGI中拯救出来了。现在，我们只需要不断地编写函数，带上URL，就可以继续Web App的开发了。

但是，Web App不仅仅是处理逻辑，展示给用户的页面也非常重要。在函数中返回一个包含HTML的字符串，简单的页面还可以，但是，想想新浪首页的6000多行的HTML，你确信能在Python的字符串中正确地写出来么？反正我是做不到。

俗话说得好，不懂前端的Python工程师不是好的产品经理。有Web开发经验的同学都明白，Web App最复杂的部分就在HTML页面。HTML不仅要正确，还要通过CSS美化，再加上复杂的JavaScript脚本来实现各种交互和动画效果。总之，生成HTML页面的难度很大。

由于在Python代码里拼字符串是不现实的，所以，模板技术出现了。

使用模板，我们需要预先准备一个HTML文档，这个HTML文档不是普通的HTML，而是嵌入了一些变量和指令，然后，根据我们传入的数据，替换后，得到最终的HTML，发送给用户：



这就是传说中的MVC：Model-View-Controller，中文名“模型-视图-控制器”。

Python处理URL的函数就是C：Controller，Controller负责业务逻辑，比如检查用户名是否存在，取出用户信息等等；

包含变量{{ name }}的模板就是V：View，View负责显示逻辑，通过简单地替换一些变量，View最终输出的就是用户看到的HTML。

MVC中的Model在哪？Model是用来传给View的，这样View在替换变量的时候，就可以从Model中取出相应的数据。

上面的例子中，Model就是一个dict：

{ 'name': 'Michael' }

只是因为Python支持关键字参数，很多Web框架允许传入关键字参数，然后，在框架内部组装出一个dict作为Model。

现在，我们把上次直接输出字符串作为HTML的例子用高端大气上档次的MVC模式改写一下：

**from** flask **import** Flask, request, render\_template

app = Flask(\_\_name\_\_)

@app.route('/', methods=['GET', 'POST'])

**def** **home**():

**return** render\_template('home.html')

@app.route('/signin', methods=['GET'])

**def** **signin\_form**():

**return** render\_template('form.html')

@app.route('/signin', methods=['POST'])

**def** **signin**():

username = request.form['username']

password = request.form['password']

**if** username=='admin' **and** password=='password':

**return** render\_template('signin-ok.html', username=username)

**return** render\_template('form.html', message='Bad username or password', username=username)

**if** \_\_name\_\_ == '\_\_main\_\_':

app.run()

Flask通过render\_template()函数来实现模板的渲染。和Web框架类似，Python的模板也有很多种。Flask默认支持的模板是[jinja2](http://jinja.pocoo.org/)，所以我们先直接安装jinja2：

$ pip install jinja2

然后，开始编写jinja2模板：

### home.html

用来显示首页的模板：

<html>

<head>

<title>Home</title>

</head>

<body>

<h1 style="font-style:italic">Home</h1>

</body>

</html>

### form.html

用来显示登录表单的模板：

<html>

<head>

<title>Please Sign In</title>

</head>

<body>

{% if message %}

<p style="color:red">{{ message }}</p>

{% endif %}

<form action="/signin" method="post">

<legend>Please sign in:</legend>

<p><input name="username" placeholder="Username" value="{{ username }}"></p>

<p><input name="password" placeholder="Password" type="password"></p>

<p><button type="submit">Sign In</button></p>

</form>

</body>

</html>

### signin-ok.html

登录成功的模板：

<html>

<head>

<title>Welcome, {{ username }}</title>

</head>

<body>

<p>Welcome, {{ username }}!</p>

</body>

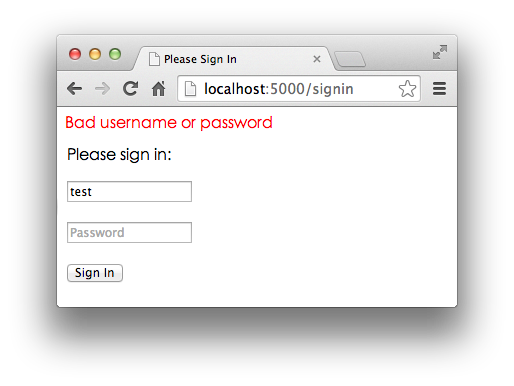
</html>

登录失败的模板呢？我们在form.html中加了一点条件判断，把form.html重用为登录失败的模板。

最后，一定要把模板放到正确的templates目录下，templates和app.py在同级目录下：

mvc-dir

启动python app.py，看看使用模板的页面效果：



通过MVC，我们在Python代码中处理M：Model和C：Controller，而V：View是通过模板处理的，这样，我们就成功地把Python代码和HTML代码最大限度地分离了。

使用模板的另一大好处是，模板改起来很方便，而且，改完保存后，刷新浏览器就能看到最新的效果，这对于调试HTML、CSS和JavaScript的前端工程师来说实在是太重要了。

在Jinja2模板中，我们用{{ name }}表示一个需要替换的变量。很多时候，还需要循环、条件判断等指令语句，在Jinja2中，用{% ... %}表示指令。

比如循环输出页码：

{% for i in page\_list %}

<a href="/page/{{ i }}">{{ i }}</a>

{% endfor %}

如果page\_list是一个list：[1, 2, 3, 4, 5]，上面的模板将输出5个超链接。

除了Jinja2，常见的模板还有：

* [Mako](http://www.makotemplates.org/)：用<% ... %>和${xxx}的一个模板；
* [Cheetah](http://www.cheetahtemplate.org/)：也是用<% ... %>和${xxx}的一个模板；
* [Django](https://www.djangoproject.com/)：Django是一站式框架，内置一个用{% ... %}和{{ xxx }}的模板。

### 小结

有了MVC，我们就分离了Python代码和HTML代码。HTML代码全部放到模板里，写起来更有效率。

### 源码参考

[app.py](https://github.com/michaelliao/learn-python3/blob/master/samples/web/mvc/app.py)

### 感觉本站内容不错，读后有收获？

#### 异步IO

阅读: 27784

在IO编程一节中，我们已经知道，CPU的速度远远快于磁盘、网络等IO。在一个线程中，CPU执行代码的速度极快，然而，一旦遇到IO操作，如读写文件、发送网络数据时，就需要等待IO操作完成，才能继续进行下一步操作。这种情况称为同步IO。

在IO操作的过程中，当前线程被挂起，而其他需要CPU执行的代码就无法被当前线程执行了。

因为一个IO操作就阻塞了当前线程，导致其他代码无法执行，所以我们必须使用多线程或者多进程来并发执行代码，为多个用户服务。每个用户都会分配一个线程，如果遇到IO导致线程被挂起，其他用户的线程不受影响。

多线程和多进程的模型虽然解决了并发问题，但是系统不能无上限地增加线程。由于系统切换线程的开销也很大，所以，一旦线程数量过多，CPU的时间就花在线程切换上了，真正运行代码的时间就少了，结果导致性能严重下降。

由于我们要解决的问题是CPU高速执行能力和IO设备的龟速严重不匹配，多线程和多进程只是解决这一问题的一种方法。

另一种解决IO问题的方法是异步IO。当代码需要执行一个耗时的IO操作时，它只发出IO指令，并不等待IO结果，然后就去执行其他代码了。一段时间后，当IO返回结果时，再通知CPU进行处理。

可以想象如果按普通顺序写出的代码实际上是没法完成异步IO的：

do\_some\_code()

f = open('/path/to/file', 'r')

r = f.read() # <== 线程停在此处等待IO操作结果

# IO操作完成后线程才能继续执行:

do\_some\_code(r)

所以，同步IO模型的代码是无法实现异步IO模型的。

异步IO模型需要一个消息循环，在消息循环中，主线程不断地重复“读取消息-处理消息”这一过程：

loop = get\_event\_loop()

**while** **True**:

event = loop.get\_event()

process\_event(event)

消息模型其实早在应用在桌面应用程序中了。一个GUI程序的主线程就负责不停地读取消息并处理消息。所有的键盘、鼠标等消息都被发送到GUI程序的消息队列中，然后由GUI程序的主线程处理。

由于GUI线程处理键盘、鼠标等消息的速度非常快，所以用户感觉不到延迟。某些时候，GUI线程在一个消息处理的过程中遇到问题导致一次消息处理时间过长，此时，用户会感觉到整个GUI程序停止响应了，敲键盘、点鼠标都没有反应。这种情况说明在消息模型中，处理一个消息必须非常迅速，否则，主线程将无法及时处理消息队列中的其他消息，导致程序看上去停止响应。

消息模型是如何解决同步IO必须等待IO操作这一问题的呢？当遇到IO操作时，代码只负责发出IO请求，不等待IO结果，然后直接结束本轮消息处理，进入下一轮消息处理过程。当IO操作完成后，将收到一条“IO完成”的消息，处理该消息时就可以直接获取IO操作结果。

在“发出IO请求”到收到“IO完成”的这段时间里，同步IO模型下，主线程只能挂起，但异步IO模型下，主线程并没有休息，而是在消息循环中继续处理其他消息。这样，在异步IO模型下，一个线程就可以同时处理多个IO请求，并且没有切换线程的操作。对于大多数IO密集型的应用程序，使用异步IO将大大提升系统的多任务处理能力。

### 感觉本站内容不错，读后有收获？

#### 协程

阅读: 42776

在学习异步IO模型前，我们先来了解协程。

协程，又称微线程，纤程。英文名Coroutine。

协程的概念很早就提出来了，但直到最近几年才在某些语言（如Lua）中得到广泛应用。

子程序，或者称为函数，在所有语言中都是层级调用，比如A调用B，B在执行过程中又调用了C，C执行完毕返回，B执行完毕返回，最后是A执行完毕。

所以子程序调用是通过栈实现的，一个线程就是执行一个子程序。

子程序调用总是一个入口，一次返回，调用顺序是明确的。而协程的调用和子程序不同。

协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。

注意，在一个子程序中中断，去执行其他子程序，不是函数调用，有点类似CPU的中断。比如子程序A、B：

**def** **A**():

print('1')

print('2')

print('3')

**def** **B**():

print('x')

print('y')

print('z')

假设由协程执行，在执行A的过程中，可以随时中断，去执行B，B也可能在执行过程中中断再去执行A，结果可能是：

1

2

x

y

3

z

但是在A中是没有调用B的，所以协程的调用比函数调用理解起来要难一些。

看起来A、B的执行有点像多线程，但协程的特点在于是一个线程执行，那和多线程比，协程有何优势？

最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。

第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

因为协程是一个线程执行，那怎么利用多核CPU呢？最简单的方法是多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

Python对协程的支持是通过generator实现的。

在generator中，我们不但可以通过for循环来迭代，还可以不断调用next()函数获取由yield语句返回的下一个值。

但是Python的yield不但可以返回一个值，它还可以接收调用者发出的参数。

来看例子：

传统的生产者-消费者模型是一个线程写消息，一个线程取消息，通过锁机制控制队列和等待，但一不小心就可能死锁。

如果改用协程，生产者生产消息后，直接通过yield跳转到消费者开始执行，待消费者执行完毕后，切换回生产者继续生产，效率极高：

**def** **consumer**():

r = ''

**while** True:

n = **yield** r

**if** **not** n:

**return**

print('[CONSUMER] Consuming %s...' % n)

r = '200 OK'

**def** **produce**(c):

c.send(None)

n = 0

**while** n < 5:

n = n + 1

print('[PRODUCER] Producing %s...' % n)

r = c.send(n)

print('[PRODUCER] Consumer return: %s' % r)

c.close()

c = consumer()

produce(c)

执行结果：

[PRODUCER] Producing 1...

[CONSUMER] Consuming 1...

[PRODUCER] Consumer return: 200 OK

[PRODUCER] Producing 2...

[CONSUMER] Consuming 2...

[PRODUCER] Consumer return: 200 OK

[PRODUCER] Producing 3...

[CONSUMER] Consuming 3...

[PRODUCER] Consumer return: 200 OK

[PRODUCER] Producing 4...

[CONSUMER] Consuming 4...

[PRODUCER] Consumer return: 200 OK

[PRODUCER] Producing 5...

[CONSUMER] Consuming 5...

[PRODUCER] Consumer return: 200 OK

注意到consumer函数是一个generator，把一个consumer传入produce后：

1. 首先调用c.send(None)启动生成器；
2. 然后，一旦生产了东西，通过c.send(n)切换到consumer执行；
3. consumer通过yield拿到消息，处理，又通过yield把结果传回；
4. produce拿到consumer处理的结果，继续生产下一条消息；
5. produce决定不生产了，通过c.close()关闭consumer，整个过程结束。

整个流程无锁，由一个线程执行，produce和consumer协作完成任务，所以称为“协程”，而非线程的抢占式多任务。

最后套用Donald Knuth的一句话总结协程的特点：

“子程序就是协程的一种特例。”

### 参考源码

[coroutine.py](https://github.com/michaelliao/learn-python3/blob/master/samples/async/coroutine.py)

### 感觉本站内容不错，读后有收获？

#### asyncio

阅读: 55802

asyncio是Python 3.4版本引入的标准库，直接内置了对异步IO的支持。

asyncio的编程模型就是一个消息循环。我们从asyncio模块中直接获取一个EventLoop的引用，然后把需要执行的协程扔到EventLoop中执行，就实现了异步IO。

用asyncio实现Hello world代码如下：

**import** asyncio

@asyncio.coroutine

**def** **hello**():

print("Hello world!")

*# 异步调用asyncio.sleep(1):*

r = **yield** **from** asyncio.sleep(1)

print("Hello again!")

*# 获取EventLoop:*

loop = asyncio.get\_event\_loop()

*# 执行coroutine*

loop.run\_until\_complete(hello())

loop.close()

@asyncio.coroutine把一个generator标记为coroutine类型，然后，我们就把这个coroutine扔到EventLoop中执行。

hello()会首先打印出Hello world!，然后，yield from语法可以让我们方便地调用另一个generator。由于asyncio.sleep()也是一个coroutine，所以线程不会等待asyncio.sleep()，而是直接中断并执行下一个消息循环。当asyncio.sleep()返回时，线程就可以从yield from拿到返回值（此处是None），然后接着执行下一行语句。

把asyncio.sleep(1)看成是一个耗时1秒的IO操作，在此期间，主线程并未等待，而是去执行EventLoop中其他可以执行的coroutine了，因此可以实现并发执行。

我们用Task封装两个coroutine试试：

**import** threading

**import** asyncio

@asyncio.coroutine

**def** **hello**():

print('Hello world! (%s)' % threading.currentThread())

**yield** **from** asyncio.sleep(1)

print('Hello again! (%s)' % threading.currentThread())

loop = asyncio.get\_event\_loop()

tasks = [hello(), hello()]

loop.run\_until\_complete(asyncio.wait(tasks))

loop.close()

观察执行过程：

Hello world! (<\_MainThread(MainThread, started 140735195337472)>)

Hello world! (<\_MainThread(MainThread, started 140735195337472)>)

(暂停约1秒)

Hello again! (<\_MainThread(MainThread, started 140735195337472)>)

Hello again! (<\_MainThread(MainThread, started 140735195337472)>)

由打印的当前线程名称可以看出，两个coroutine是由同一个线程并发执行的。

如果把asyncio.sleep()换成真正的IO操作，则多个coroutine就可以由一个线程并发执行。

我们用asyncio的异步网络连接来获取sina、sohu和163的网站首页：

**import** asyncio

@asyncio.coroutine

**def** **wget**(host):

print('wget %s...' % host)

connect = asyncio.open\_connection(host, 80)

reader, writer = **yield** **from** connect

header = 'GET / HTTP/1.0\r\nHost: %s\r\n\r\n' % host

writer.write(header.encode('utf-8'))

**yield** **from** writer.drain()

**while** True:

line = **yield** **from** reader.readline()

**if** line == b'\r\n':

**break**

print('%s header > %s' % (host, line.decode('utf-8').rstrip()))

*# Ignore the body, close the socket*

writer.close()

loop = asyncio.get\_event\_loop()

tasks = [wget(host) **for** host **in** ['www.sina.com.cn', 'www.sohu.com', 'www.163.com']]

loop.run\_until\_complete(asyncio.wait(tasks))

loop.close()

执行结果如下：

wget www.sohu.com...

wget www.sina.com.cn...

wget www.163.com...

(等待一段时间)

(打印出sohu的header)

www.sohu.com header > HTTP/1.1 200 OK

www.sohu.com header > Content-Type: text/html

...

(打印出sina的header)

www.sina.com.cn header > HTTP/1.1 200 OK

www.sina.com.cn header > Date: Wed, 20 May 2015 04:56:33 GMT

...

(打印出163的header)

www.163.com header > HTTP/1.0 302 Moved Temporarily

www.163.com header > Server: Cdn Cache Server V2.0

...

可见3个连接由一个线程通过coroutine并发完成。

### 小结

asyncio提供了完善的异步IO支持；

异步操作需要在coroutine中通过yield from完成；

多个coroutine可以封装成一组Task然后并发执行。

### 参考源码

[async\_hello.py](https://github.com/michaelliao/learn-python3/blob/master/samples/async/async_hello.py)

[async\_wget.py](https://github.com/michaelliao/learn-python3/blob/master/samples/async/async_wget.py)

### 感觉本站内容不错，读后有收获？

#### async/await

阅读: 16982

用asyncio提供的@asyncio.coroutine可以把一个generator标记为coroutine类型，然后在coroutine内部用yield from调用另一个coroutine实现异步操作。

为了简化并更好地标识异步IO，从Python 3.5开始引入了新的语法async和await，可以让coroutine的代码更简洁易读。

请注意，async和await是针对coroutine的新语法，要使用新的语法，只需要做两步简单的替换：

1. 把@asyncio.coroutine替换为async；
2. 把yield from替换为await。

让我们对比一下上一节的代码：

@asyncio.coroutine

**def** **hello**():

print("Hello world!")

r = **yield** **from** asyncio.sleep(1)

print("Hello again!")

用新语法重新编写如下：

async **def** **hello**():

print("Hello world!")

r = await asyncio.sleep(1)

print("Hello again!")

剩下的代码保持不变。

### 小结

Python从3.5版本开始为asyncio提供了async和await的新语法；

注意新语法只能用在Python 3.5以及后续版本，如果使用3.4版本，则仍需使用上一节的方案。

### 练习

将上一节的异步获取sina、sohu和163的网站首页源码用新语法重写并运行。

### 参考源码

[async\_hello2.py](https://github.com/michaelliao/learn-python3/blob/master/samples/async/async_hello2.py)

[async\_wget2.py](https://github.com/michaelliao/learn-python3/blob/master/samples/async/async_wget2.py)

### 感觉本站内容不错，读后有收获？

#### aiohttp

阅读: 23194

asyncio可以实现单线程并发IO操作。如果仅用在客户端，发挥的威力不大。如果把asyncio用在服务器端，例如Web服务器，由于HTTP连接就是IO操作，因此可以用单线程+coroutine实现多用户的高并发支持。

asyncio实现了TCP、UDP、SSL等协议，aiohttp则是基于asyncio实现的HTTP框架。

我们先安装aiohttp：

pip install aiohttp

然后编写一个HTTP服务器，分别处理以下URL：

* / - 首页返回b'<h1>Index</h1>'；
* /hello/{name} - 根据URL参数返回文本hello, %s!。

代码如下：

**import** asyncio

**from** aiohttp **import** web

async **def** **index**(request):

await asyncio.sleep(0.5)

**return** web.Response(body=b'<h1>Index</h1>')

async **def** **hello**(request):

await asyncio.sleep(0.5)

text = '<h1>hello, %s!</h1>' % request.match\_info['name']

**return** web.Response(body=text.encode('utf-8'))

async **def** **init**(loop):

app = web.Application(loop=loop)

app.router.add\_route('GET', '/', index)

app.router.add\_route('GET', '/hello/{name}', hello)

srv = await loop.create\_server(app.make\_handler(), '127.0.0.1', 8000)

print('Server started at http://127.0.0.1:8000...')

**return** srv

loop = asyncio.get\_event\_loop()

loop.run\_until\_complete(init(loop))

loop.run\_forever()

注意aiohttp的初始化函数init()也是一个coroutine，loop.create\_server()则利用asyncio创建TCP服务。

### 参考源码

[aio\_web.py](https://github.com/michaelliao/learn-python3/blob/master/samples/async/aio_web.py)

### 感觉本站内容不错，读后有收获？

#### 实战

阅读: 10606

看完了教程，是不是有这么一种感觉：看的时候觉得很简单，照着教程敲代码也没啥大问题。

于是准备开始独立写代码，就发现不知道从哪开始下手了。

这种情况是完全正常的。好比学写作文，学的时候觉得简单，写的时候就无从下笔了。

虽然这个教程是面向小白的零基础Python教程，但是我们的目标不是学到60分，而是学到90分。

所以，用Python写一个真正的Web App吧！

### 目标

我们设定的实战目标是一个Blog网站，包含日志、用户和评论3大部分。

很多童鞋会想，这是不是太简单了？

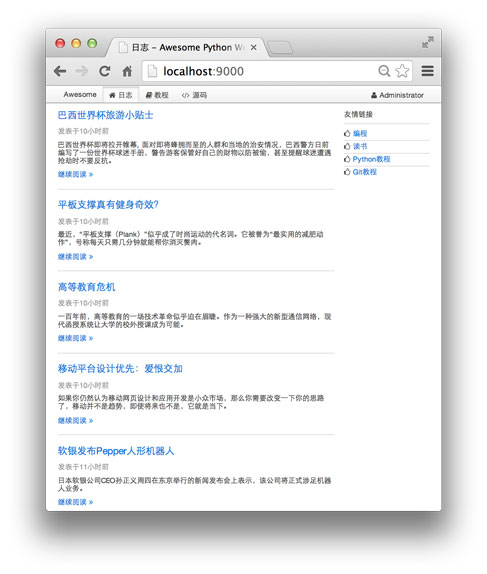
比如[webpy.org](http://webpy.org/src/blog/0.3)上就提供了一个Blog的例子，目测也就100行代码。

但是，这样的页面：

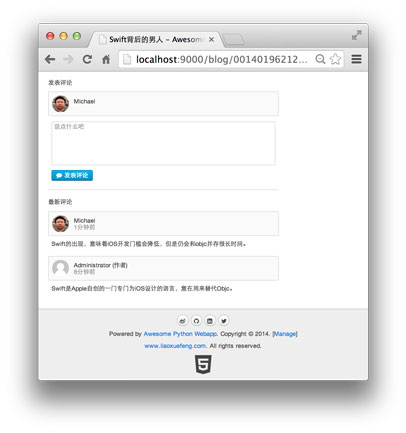


你拿得出手么？

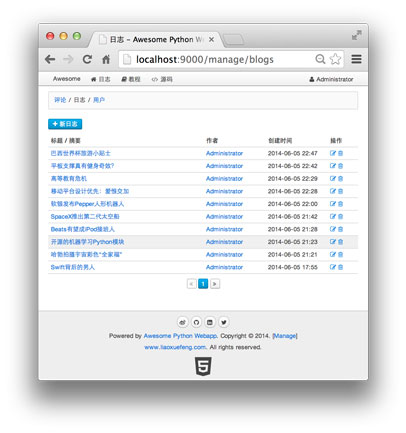
我们要写出用户真正看得上眼的页面，首页长得像这样：



评论区：



还有极其强大的后台管理页面：



是不是一下子变得高端大气上档次了？

### 项目名称

必须是高端大气上档次的名称，命名为awesome-python3-webapp。

### 项目计划

项目计划开发周期为16天。每天，你需要完成教程中的内容。如果你觉得编写代码难度实在太大，可以参考一下当天在GitHub上的代码。

第N天的代码在https://github.com/michaelliao/awesome-python3-webapp/tree/day-N上。比如第1天就是：

<https://github.com/michaelliao/awesome-python3-webapp/tree/day-01>

以此类推。

要预览awesome-python3-webapp的最终页面效果，请猛击：

### [awesome.liaoxuefeng.com](http://awesome.liaoxuefeng.com/)

### 感觉本站内容不错，读后有收获？

#### Day 1 - 搭建开发环境

阅读: 184565

### 搭建开发环境

首先，确认系统安装的Python版本是3.5.x：

$ python3 --version

Python 3.5.1

然后，用pip安装开发Web App需要的第三方库：

异步框架aiohttp：

$pip3 install aiohttp

前端模板引擎jinja2：

$ pip3 install jinja2

MySQL 5.x数据库，从[官方网站](http://dev.mysql.com/downloads/mysql/5.6.html)下载并安装，安装完毕后，请务必牢记root口令。为避免遗忘口令，建议直接把root口令设置为password；

MySQL的Python异步驱动程序aiomysql：

$ pip3 install aiomysql

### 项目结构

选择一个工作目录，然后，我们建立如下的目录结构：

awesome-python3-webapp/ <*-- 根目录*

|

+- backup/ <*-- 备份目录*

|

+- conf/ <*-- 配置文件*

|

+- dist/ <*-- 打包目录*

|

+- www/ <*-- Web目录，存放.py文件*

| |

| +- static/ <*-- 存放静态文件*

| |

| +- templates/ <*-- 存放模板文件*

|

+- ios/ <*-- 存放iOS App工程*

|

+- LICENSE <*-- 代码LICENSE*

创建好项目的目录结构后，建议同时建立git仓库并同步至GitHub，保证代码修改的安全。

要了解git和GitHub的用法，请移步[Git教程](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000)。

### 开发工具

自备，推荐用Sublime Text，请参考[使用文本编辑器](http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/0014316399410395f704750ee9440228135925a6ca1dad8000)。

### 参考源码

[day-01](https://github.com/michaelliao/awesome-python3-webapp/tree/day-01)

### 感觉本站内容不错，读后有收获？

#### Day 2 - 编写Web App骨架

阅读: 125508

由于我们的Web App建立在asyncio的基础上，因此用aiohttp写一个基本的app.py：

**import** logging; logging.basicConfig(level=logging.INFO)

**import** asyncio, os, json, time

**from** datetime **import** datetime

**from** aiohttp **import** web

**def** **index**(request):

**return** web.Response(body=b'<h1>Awesome</h1>')

@asyncio.coroutine

**def** **init**(loop):

app = web.Application(loop=loop)

app.router.add\_route('GET', '/', index)

srv = **yield** **from** loop.create\_server(app.make\_handler(), '127.0.0.1', 9000)

logging.info('server started at http://127.0.0.1:9000...')

**return** srv

loop = asyncio.get\_event\_loop()

loop.run\_until\_complete(init(loop))

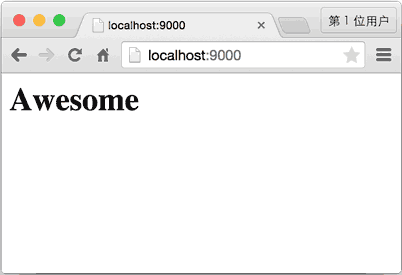
loop.run\_forever()

运行python app.py，Web App将在9000端口监听HTTP请求，并且对首页/进行响应：

$ python3 app.py

INFO:root:server started at http:*//127.0.0.1:9000...*

这里我们简单地返回一个Awesome字符串，在浏览器中可以看到效果：



这说明我们的Web App骨架已经搭好了，可以进一步往里面添加更多的东西。

### 参考源码

[day-02](https://github.com/michaelliao/awesome-python3-webapp/tree/day-02)

### 感觉本站内容不错，读后有收获？

#### Day 3 - 编写ORM

阅读: 125419

在一个Web App中，所有数据，包括用户信息、发布的日志、评论等，都存储在数据库中。在awesome-python3-webapp中，我们选择MySQL作为数据库。

Web App里面有很多地方都要访问数据库。访问数据库需要创建数据库连接、游标对象，然后执行SQL语句，最后处理异常，清理资源。这些访问数据库的代码如果分散到各个函数中，势必无法维护，也不利于代码复用。

所以，我们要首先把常用的SELECT、INSERT、UPDATE和DELETE操作用函数封装起来。

由于Web框架使用了基于asyncio的aiohttp，这是基于协程的异步模型。在协程中，不能调用普通的同步IO操作，因为所有用户都是由一个线程服务的，协程的执行速度必须非常快，才能处理大量用户的请求。而耗时的IO操作不能在协程中以同步的方式调用，否则，等待一个IO操作时，系统无法响应任何其他用户。

这就是异步编程的一个原则：一旦决定使用异步，则系统每一层都必须是异步，“开弓没有回头箭”。

幸运的是aiomysql为MySQL数据库提供了异步IO的驱动。

### 创建连接池

我们需要创建一个全局的连接池，每个HTTP请求都可以从连接池中直接获取数据库连接。使用连接池的好处是不必频繁地打开和关闭数据库连接，而是能复用就尽量复用。

连接池由全局变量\_\_pool存储，缺省情况下将编码设置为utf8，自动提交事务：

@asyncio.coroutine

**def** **create\_pool**(loop, \*\*kw):

logging.info('create database connection pool...')

**global** \_\_pool

\_\_pool = **yield** **from** aiomysql.create\_pool(

host=kw.get('host', 'localhost'),

port=kw.get('port', 3306),

user=kw['user'],

password=kw['password'],

db=kw['db'],

charset=kw.get('charset', 'utf8'),

autocommit=kw.get('autocommit', True),

maxsize=kw.get('maxsize', 10),

minsize=kw.get('minsize', 1),

loop=loop

)

### Select

要执行SELECT语句，我们用select函数执行，需要传入SQL语句和SQL参数：

@asyncio.coroutine

**def** **select**(sql, args, size=None):

log(sql, args)

**global** \_\_pool

**with** (**yield** **from** \_\_pool) **as** conn:

cur = **yield** **from** conn.cursor(aiomysql.DictCursor)

**yield** **from** cur.execute(sql.replace('?', '%s'), args **or** ())

**if** size:

rs = **yield** **from** cur.fetchmany(size)

**else**:

rs = **yield** **from** cur.fetchall()

**yield** **from** cur.close()

logging.info('rows returned: %s' % len(rs))

**return** rs

SQL语句的占位符是?，而MySQL的占位符是%s，select()函数在内部自动替换。注意要始终坚持使用带参数的SQL，而不是自己拼接SQL字符串，这样可以防止SQL注入攻击。

注意到yield from将调用一个子协程（也就是在一个协程中调用另一个协程）并直接获得子协程的返回结果。

如果传入size参数，就通过fetchmany()获取最多指定数量的记录，否则，通过fetchall()获取所有记录。

### Insert, Update, Delete

要执行INSERT、UPDATE、DELETE语句，可以定义一个通用的execute()函数，因为这3种SQL的执行都需要相同的参数，以及返回一个整数表示影响的行数：

@asyncio.coroutine

**def** **execute**(sql, args):

log(sql)

**with** (**yield** **from** \_\_pool) **as** conn:

**try**:

cur = **yield** **from** conn.cursor()

**yield** **from** cur.execute(sql.replace('?', '%s'), args)

affected = cur.rowcount

**yield** **from** cur.close()

**except** BaseException **as** e:

**raise**

**return** affected

execute()函数和select()函数所不同的是，cursor对象不返回结果集，而是通过rowcount返回结果数。

### ORM

有了基本的select()和execute()函数，我们就可以开始编写一个简单的ORM了。

设计ORM需要从上层调用者角度来设计。

我们先考虑如何定义一个User对象，然后把数据库表users和它关联起来。

**from** orm **import** Model, StringField, IntegerField

**class User(Model):**

\_\_table\_\_ = 'users'

id = IntegerField(primary\_key=True)

name = StringField()

注意到定义在User类中的\_\_table\_\_、id和name是类的属性，不是实例的属性。所以，在类级别上定义的属性用来描述User对象和表的映射关系，而实例属性必须通过\_\_init\_\_()方法去初始化，所以两者互不干扰：

# 创建实例:

user = User(id=123, name='Michael')

# 存入数据库:

user.**insert**()

# 查询所有**User**对象:

users = **User**.findAll()

### 定义Model

首先要定义的是所有ORM映射的基类Model：

**class Model(dict, metaclass=ModelMetaclass):**

**def** **\_\_init\_\_**(self, \*\*kw):

super(Model, self).\_\_init\_\_(\*\*kw)

**def** **\_\_getattr\_\_**(self, key):

**try**:

**return** self[key]

**except** KeyError:

**raise** AttributeError(r"'Model' object has no attribute '%s'" % key)

**def** **\_\_setattr\_\_**(self, key, value):

self[key] = value

**def** **getValue**(self, key):

**return** getattr(self, key, None)

**def** **getValueOrDefault**(self, key):

value = getattr(self, key, None)

**if** value **is** None:

field = self.\_\_mappings\_\_[key]

**if** field.default **is** **not** None:

value = field.default() **if** callable(field.default) **else** field.default

logging.debug('using default value for %s: %s' % (key, str(value)))

setattr(self, key, value)

**return** value

Model从dict继承，所以具备所有dict的功能，同时又实现了特殊方法\_\_getattr\_\_()和\_\_setattr\_\_()，因此又可以像引用普通字段那样写：

>>> user['id']

123

>>> user.id

123

以及Field和各种Field子类：

**class Field(object):**

**def** **\_\_init\_\_**(self, name, column\_type, primary\_key, default):

self.name = name

self.column\_type = column\_type

self.primary\_key = primary\_key

self.default = default

**def** **\_\_str\_\_**(self):

**return** '<%s, %s:%s>' % (self.\_\_class\_\_.\_\_name\_\_, self.column\_type, self.name)

映射varchar的StringField：

**class StringField(Field):**

**def** **\_\_init\_\_**(self, name=None, primary\_key=False, default=None, ddl='varchar(100)'):

super().\_\_init\_\_(name, ddl, primary\_key, default)

注意到Model只是一个基类，如何将具体的子类如User的映射信息读取出来呢？答案就是通过metaclass：ModelMetaclass：

**class ModelMetaclass(type):**

**def** **\_\_new\_\_**(cls, name, bases, attrs):

*# 排除Model类本身:*

**if** name=='Model':

**return** type.\_\_new\_\_(cls, name, bases, attrs)

*# 获取table名称:*

tableName = attrs.get('\_\_table\_\_', None) **or** name

logging.info('found model: %s (table: %s)' % (name, tableName))

*# 获取所有的Field和主键名:*

mappings = dict()

fields = []

primaryKey = None

**for** k, v **in** attrs.items():

**if** isinstance(v, Field):

logging.info(' found mapping: %s ==> %s' % (k, v))

mappings[k] = v

**if** v.primary\_key:

*# 找到主键:*

**if** primaryKey:

**raise** RuntimeError('Duplicate primary key for field: %s' % k)

primaryKey = k

**else**:

fields.append(k)

**if** **not** primaryKey:

**raise** RuntimeError('Primary key not found.')

**for** k **in** mappings.keys():

attrs.pop(k)

escaped\_fields = list(map(**lambda** f: '`%s`' % f, fields))

attrs['\_\_mappings\_\_'] = mappings *# 保存属性和列的映射关系*

attrs['\_\_table\_\_'] = tableName

attrs['\_\_primary\_key\_\_'] = primaryKey *# 主键属性名*

attrs['\_\_fields\_\_'] = fields *# 除主键外的属性名*

*# 构造默认的SELECT, INSERT, UPDATE和DELETE语句:*

attrs['\_\_select\_\_'] = 'select `%s`, %s from `%s`' % (primaryKey, ', '.join(escaped\_fields), tableName)

attrs['\_\_insert\_\_'] = 'insert into `%s` (%s, `%s`) values (%s)' % (tableName, ', '.join(escaped\_fields), primaryKey, create\_args\_string(len(escaped\_fields) + 1))

attrs['\_\_update\_\_'] = 'update `%s` set %s where `%s`=?' % (tableName, ', '.join(map(**lambda** f: '`%s`=?' % (mappings.get(f).name **or** f), fields)), primaryKey)

attrs['\_\_delete\_\_'] = 'delete from `%s` where `%s`=?' % (tableName, primaryKey)

**return** type.\_\_new\_\_(cls, name, bases, attrs)

这样，任何继承自Model的类（比如User），会自动通过ModelMetaclass扫描映射关系，并存储到自身的类属性如\_\_table\_\_、\_\_mappings\_\_中。

然后，我们往Model类添加class方法，就可以让所有子类调用class方法：

**class Model(dict):**

...

@classmethod

@asyncio.coroutine

**def** **find**(cls, pk):

' find object by primary key. '

rs = **yield** **from** select('%s where `%s`=?' % (cls.\_\_select\_\_, cls.\_\_primary\_key\_\_), [pk], 1)

**if** len(rs) == 0:

**return** None

**return** cls(\*\*rs[0])

User类现在就可以通过类方法实现主键查找：

user = **yield** **from** User.find('123')

往Model类添加实例方法，就可以让所有子类调用实例方法：

**class Model(dict):**

...

@asyncio.coroutine

**def** **save**(self):

args = list(map(self.getValueOrDefault, self.\_\_fields\_\_))

args.append(self.getValueOrDefault(self.\_\_primary\_key\_\_))

rows = **yield** **from** execute(self.\_\_insert\_\_, args)

**if** rows != 1:

logging.warn('failed to insert record: affected rows: %s' % rows)

这样，就可以把一个User实例存入数据库：

user = User(id=123, name='Michael')

**yield** **from** user.save()

最后一步是完善ORM，对于查找，我们可以实现以下方法：

* findAll() - 根据WHERE条件查找；
* findNumber() - 根据WHERE条件查找，但返回的是整数，适用于select count(\*)类型的SQL。

以及update()和remove()方法。

所有这些方法都必须用@asyncio.coroutine装饰，变成一个协程。

调用时需要特别注意：

user.save()

没有任何效果，因为调用save()仅仅是创建了一个协程，并没有执行它。一定要用：

**yield** **from** user.save()

才真正执行了INSERT操作。

最后看看我们实现的ORM模块一共多少行代码？累计不到300多行。用Python写一个ORM是不是很容易呢？

### 参考源码

[day-03](https://github.com/michaelliao/awesome-python3-webapp/tree/day-03)

### 感觉本站内容不错，读后有收获？

#### Day 4 - 编写Model

阅读: 26618

有了ORM，我们就可以把Web App需要的3个表用Model表示出来：

**import** time, uuid

**from** orm **import** Model, StringField, BooleanField, FloatField, TextField

**def** **next\_id**():

**return** '%015d%s000' % (int(time.time() \* 1000), uuid.uuid4().hex)

**class User(Model):**

\_\_table\_\_ = 'users'

id = StringField(primary\_key=True, default=next\_id, ddl='varchar(50)')

email = StringField(ddl='varchar(50)')

passwd = StringField(ddl='varchar(50)')

admin = BooleanField()

name = StringField(ddl='varchar(50)')

image = StringField(ddl='varchar(500)')

created\_at = FloatField(default=time.time)

**class Blog(Model):**

\_\_table\_\_ = 'blogs'

id = StringField(primary\_key=True, default=next\_id, ddl='varchar(50)')

user\_id = StringField(ddl='varchar(50)')

user\_name = StringField(ddl='varchar(50)')

user\_image = StringField(ddl='varchar(500)')

name = StringField(ddl='varchar(50)')

summary = StringField(ddl='varchar(200)')

content = TextField()

created\_at = FloatField(default=time.time)

**class Comment(Model):**

\_\_table\_\_ = 'comments'

id = StringField(primary\_key=True, default=next\_id, ddl='varchar(50)')

blog\_id = StringField(ddl='varchar(50)')

user\_id = StringField(ddl='varchar(50)')

user\_name = StringField(ddl='varchar(50)')

user\_image = StringField(ddl='varchar(500)')

content = TextField()

created\_at = FloatField(default=time.time)

在编写ORM时，给一个Field增加一个default参数可以让ORM自己填入缺省值，非常方便。并且，缺省值可以作为函数对象传入，在调用save()时自动计算。

例如，主键id的缺省值是函数next\_id，创建时间created\_at的缺省值是函数time.time，可以自动设置当前日期和时间。

日期和时间用float类型存储在数据库中，而不是datetime类型，这么做的好处是不必关心数据库的时区以及时区转换问题，排序非常简单，显示的时候，只需要做一个float到str的转换，也非常容易。

### 初始化数据库表

如果表的数量很少，可以手写创建表的SQL脚本：

*-- schema.sql*

**drop** database if **exists** awesome;

**create** database awesome;

use awesome;

**grant** **select**, **insert**, **update**, **delete** **on** awesome.\* **to** 'www-data'@'localhost' identified **by** 'www-data';

**create** **table** users (

`id` **varchar**(50) **not** **null**,

`email` **varchar**(50) **not** **null**,

`passwd` **varchar**(50) **not** **null**,

`admin` bool **not** **null**,

`name` **varchar**(50) **not** **null**,

`image` **varchar**(500) **not** **null**,

`created\_at` **real** **not** **null**,

**unique** **key** `idx\_email` (`email`),

**key** `idx\_created\_at` (`created\_at`),

**primary** **key** (`id`)

) engine=innodb **default** charset=utf8;

**create** **table** blogs (

`id` **varchar**(50) **not** **null**,

`user\_id` **varchar**(50) **not** **null**,

`user\_name` **varchar**(50) **not** **null**,

`user\_image` **varchar**(500) **not** **null**,

`name` **varchar**(50) **not** **null**,

`summary` **varchar**(200) **not** **null**,

`content` mediumtext **not** **null**,

`created\_at` **real** **not** **null**,

**key** `idx\_created\_at` (`created\_at`),

**primary** **key** (`id`)

) engine=innodb **default** charset=utf8;

**create** **table** comments (

`id` **varchar**(50) **not** **null**,

`blog\_id` **varchar**(50) **not** **null**,

`user\_id` **varchar**(50) **not** **null**,

`user\_name` **varchar**(50) **not** **null**,

`user\_image` **varchar**(500) **not** **null**,

`content` mediumtext **not** **null**,

`created\_at` **real** **not** **null**,

**key** `idx\_created\_at` (`created\_at`),

**primary** **key** (`id`)

) engine=innodb **default** charset=utf8;

如果表的数量很多，可以从Model对象直接通过脚本自动生成SQL脚本，使用更简单。

把SQL脚本放到MySQL命令行里执行：

$ mysql -u root -p < schema.sql

我们就完成了数据库表的初始化。

### 编写数据访问代码

接下来，就可以真正开始编写代码操作对象了。比如，对于User对象，我们就可以做如下操作：

**import** orm

**from** models **import** User, Blog, Comment

**def** **test**():

**yield** **from** orm.create\_pool(user='www-data', password='www-data', database='awesome')

u = User(name='Test', email='test@example.com', passwd='1234567890', image='about:blank')

**yield** **from** u.save()

**for** x **in** test():

**pass**

可以在MySQL客户端命令行查询，看看数据是不是正常存储到MySQL里面了。

### 参考源码

[day-04](https://github.com/michaelliao/awesome-python3-webapp/tree/day-04)

### 感觉本站内容不错，读后有收获？

#### Day 5 - 编写Web框架

阅读: 34248

在正式开始Web开发前，我们需要编写一个Web框架。

aiohttp已经是一个Web框架了，为什么我们还需要自己封装一个？

原因是从使用者的角度来说，aiohttp相对比较底层，编写一个URL的处理函数需要这么几步：

第一步，编写一个用@asyncio.coroutine装饰的函数：

@asyncio.coroutine

**def** **handle\_url\_xxx**(request):

**pass**

第二步，传入的参数需要自己从request中获取：

url\_param = request.match\_info['key']

query\_params = parse\_qs(request.query\_string)

最后，需要自己构造Response对象：

text = render('template', data)

**return** web.Response(text.encode('utf-8'))

这些重复的工作可以由框架完成。例如，处理带参数的URL/blog/{id}可以这么写：

@get('/blog/{id}')

**def** **get\_blog**(id):

**pass**

处理query\_string参数可以通过关键字参数\*\*kw或者命名关键字参数接收：

@get('/api/comments')

**def** **api\_comments**(\*, page='1'):

**pass**

对于函数的返回值，不一定是web.Response对象，可以是str、bytes或dict。

如果希望渲染模板，我们可以这么返回一个dict：

**return** {

'\_\_template\_\_': 'index.html',

'data': '...'

}

因此，Web框架的设计是完全从使用者出发，目的是让使用者编写尽可能少的代码。

编写简单的函数而非引入request和web.Response还有一个额外的好处，就是可以单独测试，否则，需要模拟一个request才能测试。

### @get和@post

要把一个函数映射为一个URL处理函数，我们先定义@get()：

**def** **get**(path):

'''

Define decorator @get('/path')

'''

**def** **decorator**(func):

@functools.wraps(func)

**def** **wrapper**(\*args, \*\*kw):

**return** func(\*args, \*\*kw)

wrapper.\_\_method\_\_ = 'GET'

wrapper.\_\_route\_\_ = path

**return** wrapper

**return** decorator

这样，一个函数通过@get()的装饰就附带了URL信息。

@post与@get定义类似。

### 定义RequestHandler

URL处理函数不一定是一个coroutine，因此我们用RequestHandler()来封装一个URL处理函数。

RequestHandler是一个类，由于定义了\_\_call\_\_()方法，因此可以将其实例视为函数。

RequestHandler目的就是从URL函数中分析其需要接收的参数，从request中获取必要的参数，调用URL函数，然后把结果转换为web.Response对象，这样，就完全符合aiohttp框架的要求：

**class RequestHandler(object):**

**def** **\_\_init\_\_**(self, app, fn):

self.\_app = app

self.\_func = fn

...

@asyncio.coroutine

**def** **\_\_call\_\_**(self, request):

kw = ... 获取参数

r = **yield** **from** self.\_func(\*\*kw)

**return** r

再编写一个add\_route函数，用来注册一个URL处理函数：

**def** **add\_route**(app, fn):

method = getattr(fn, '\_\_method\_\_', None)

path = getattr(fn, '\_\_route\_\_', None)

**if** path **is** None **or** method **is** None:

**raise** ValueError('@get or @post not defined in %s.' % str(fn))

**if** **not** asyncio.iscoroutinefunction(fn) **and** **not** inspect.isgeneratorfunction(fn):

fn = asyncio.coroutine(fn)

logging.info('add route %s %s => %s(%s)' % (method, path, fn.\_\_name\_\_, ', '.join(inspect.signature(fn).parameters.keys())))

app.router.add\_route(method, path, RequestHandler(app, fn))

最后一步，把很多次add\_route()注册的调用：

add\_route(app, handles.index)

add\_route(app, handles.blog)

add\_route(app, handles.create\_comment)

...

变成自动扫描：

# 自动把**handler**模块的所有符合条件的函数注册了:

add\_routes(app, 'handlers')

add\_routes()定义如下：

**def** **add\_routes**(app, module\_name):

n = module\_name.rfind('.')

**if** n == (-1):

mod = \_\_import\_\_(module\_name, globals(), locals())

**else**:

name = module\_name[n+1:]

mod = getattr(\_\_import\_\_(module\_name[:n], globals(), locals(), [name]), name)

**for** attr **in** dir(mod):

**if** attr.startswith('\_'):

**continue**

fn = getattr(mod, attr)

**if** callable(fn):

method = getattr(fn, '\_\_method\_\_', None)

path = getattr(fn, '\_\_route\_\_', None)

**if** method **and** path:

add\_route(app, fn)

最后，在app.py中加入middleware、jinja2模板和自注册的支持：

app = web.Application(loop=loop, middlewares=[

logger\_factory, response\_factory

])

init\_jinja2(app, filters=dict(datetime=datetime\_filter))

add\_routes(app, 'handlers')

add\_static(app)

### middleware

middleware是一种拦截器，一个URL在被某个函数处理前，可以经过一系列的middleware的处理。

一个middleware可以改变URL的输入、输出，甚至可以决定不继续处理而直接返回。middleware的用处就在于把通用的功能从每个URL处理函数中拿出来，集中放到一个地方。例如，一个记录URL日志的logger可以简单定义如下：

@asyncio.coroutine

**def** **logger\_factory**(app, handler):

@asyncio.coroutine

**def** **logger**(request):

*# 记录日志:*

logging.info('Request: %s %s' % (request.method, request.path))

*# 继续处理请求:*

**return** (**yield** **from** handler(request))

**return** logger

而response这个middleware把返回值转换为web.Response对象再返回，以保证满足aiohttp的要求：

@asyncio.coroutine

**def** **response\_factory**(app, handler):

@asyncio.coroutine

**def** **response**(request):

*# 结果:*

r = **yield** **from** handler(request)

**if** isinstance(r, web.StreamResponse):

**return** r

**if** isinstance(r, bytes):

resp = web.Response(body=r)

resp.content\_type = 'application/octet-stream'

**return** resp

**if** isinstance(r, str):

resp = web.Response(body=r.encode('utf-8'))

resp.content\_type = 'text/html;charset=utf-8'

**return** resp

**if** isinstance(r, dict):

...

有了这些基础设施，我们就可以专注地往handlers模块不断添加URL处理函数了，可以极大地提高开发效率。

### 参考源码

[day-05](https://github.com/michaelliao/awesome-python3-webapp/tree/day-05)

### 感觉本站内容不错，读后有收获？

#### Day 6 - 编写配置文件

阅读: 2550

有了Web框架和ORM框架，我们就可以开始装配App了。

通常，一个Web App在运行时都需要读取配置文件，比如数据库的用户名、口令等，在不同的环境中运行时，Web App可以通过读取不同的配置文件来获得正确的配置。

由于Python本身语法简单，完全可以直接用Python源代码来实现配置，而不需要再解析一个单独的.properties或者.yaml等配置文件。

默认的配置文件应该完全符合本地开发环境，这样，无需任何设置，就可以立刻启动服务器。

我们把默认的配置文件命名为config\_default.py：

*# config\_default.py*

configs = {

'db': {

'host': '127.0.0.1',

'port': 3306,

'user': 'www-data',

'password': 'www-data',

'database': 'awesome'

},

'session': {

'secret': 'AwEsOmE'

}

}

上述配置文件简单明了。但是，如果要部署到服务器时，通常需要修改数据库的host等信息，直接修改config\_default.py不是一个好办法，更好的方法是编写一个config\_override.py，用来覆盖某些默认设置：

*# config\_override.py*

configs = {

'db': {

'host': '192.168.0.100'

}

}

把config\_default.py作为开发环境的标准配置，把config\_override.py作为生产环境的标准配置，我们就可以既方便地在本地开发，又可以随时把应用部署到服务器上。

应用程序读取配置文件需要优先从config\_override.py读取。为了简化读取配置文件，可以把所有配置读取到统一的config.py中：

*# config.py*

configs = config\_default.configs

**try**:

**import** config\_override

configs = merge(configs, config\_override.configs)

**except** ImportError:

**pass**

这样，我们就完成了App的配置。

### 参考源码

[day-06](https://github.com/michaelliao/awesome-python3-webapp/tree/day-06)

### 感觉本站内容不错，读后有收获？

#### Day 7 - 编写MVC

阅读: 11271

现在，ORM框架、Web框架和配置都已就绪，我们可以开始编写一个最简单的MVC，把它们全部启动起来。

通过Web框架的@get和ORM框架的Model支持，可以很容易地编写一个处理首页URL的函数：

@get('/')

**def** **index**(request):

users = **yield** **from** User.findAll()

**return** {

'\_\_template\_\_': 'test.html',

'users': users

}

'\_\_template\_\_'指定的模板文件是test.html，其他参数是传递给模板的数据，所以我们在模板的根目录templates下创建test.html：

**<!DOCTYPE html>**

<html>

<head>

<meta charset="utf-8" />

<title>Test users - Awesome Python Webapp</title>

</head>

<body>

<h1>All users</h1>

{% for u in users %}

<p>{{ u.name }} / {{ u.email }}</p>

{% endfor %}

</body>

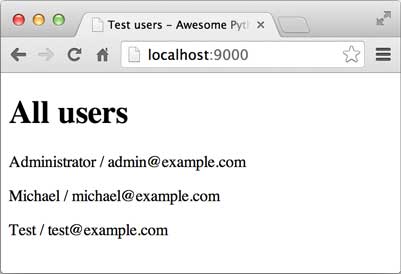
</html>

接下来，如果一切顺利，可以用命令行启动Web服务器：

$ python3 app.py

然后，在浏览器中访问http://localhost:9000/。

如果数据库的users表什么内容也没有，你就无法在浏览器中看到循环输出的内容。可以自己在MySQL的命令行里给users表添加几条记录，然后再访问：



### 参考源码

[day-07](https://github.com/michaelliao/awesome-python3-webapp/tree/day-07)

### 感觉本站内容不错，读后有收获？

#### Day 8 - 构建前端

阅读: 11331

虽然我们跑通了一个最简单的MVC，但是页面效果肯定不会让人满意。

对于复杂的HTML前端页面来说，我们需要一套基础的CSS框架来完成页面布局和基本样式。另外，jQuery作为操作DOM的JavaScript库也必不可少。

从零开始写CSS不如直接从一个已有的功能完善的CSS框架开始。有很多CSS框架可供选择。我们这次选择[uikit](http://getuikit.com/)这个强大的CSS框架。它具备完善的响应式布局，漂亮的UI，以及丰富的HTML组件，让我们能轻松设计出美观而简洁的页面。

可以从[uikit首页](http://getuikit.com/)下载打包的资源文件。

所有的静态资源文件我们统一放到www/static目录下，并按照类别归类：

**static**/

+- css/

| +- addons/

| | +- uikit.addons.min.css

| | +- uikit.almost-flat.addons.min.css

| | +- uikit.gradient.addons.min.css

| +- awesome.css

| +- uikit.almost-flat.addons.min.css

| +- uikit.gradient.addons.min.css

| +- uikit.min.css

+- fonts/

| +- fontawesome-webfont.eot

| +- fontawesome-webfont.ttf

| +- fontawesome-webfont.woff

| +- FontAwesome.otf

+- js/

+- awesome.js

+- html5.js

+- jquery.min.js

+- uikit.min.js

由于前端页面肯定不止首页一个页面，每个页面都有相同的页眉和页脚。如果每个页面都是独立的HTML模板，那么我们在修改页眉和页脚的时候，就需要把每个模板都改一遍，这显然是没有效率的。

常见的模板引擎已经考虑到了页面上重复的HTML部分的复用问题。有的模板通过include把页面拆成三部分：

<html>

<% include file="inc\_header.html" %>

<% include file="index\_body.html" %>

<% include file="inc\_footer.html" %>

</html>

这样，相同的部分inc\_header.html和inc\_footer.html就可以共享。

但是include方法不利于页面整体结构的维护。jinjia2的模板还有另一种“继承”方式，实现模板的复用更简单。

“继承”模板的方式是通过编写一个“父模板”，在父模板中定义一些可替换的block（块）。然后，编写多个“子模板”，每个子模板都可以只替换父模板定义的block。比如，定义一个最简单的父模板：

*<!-- base.html -->*

<html>

<head>

<title>{% block title%} 这里定义了一个名为title的block {% endblock %}</title>

</head>

<body>

{% block content %} 这里定义了一个名为content的block {% endblock %}

</body>

</html>

对于子模板a.html，只需要把父模板的title和content替换掉：

{% extends 'base.html' %}

{% block title %} A {% endblock %}

{% block content %}

<h1>Chapter A</h1>

<p>blablabla...</p>

{% endblock %}

对于子模板b.html，如法炮制：

{% extends 'base.html' %}

{% block title %} B {% endblock %}

{% block content %}

<h1>Chapter B</h1>

<ul>

<li>list 1</li>

<li>list 2</li>

</ul>

{% endblock %}

这样，一旦定义好父模板的整体布局和CSS样式，编写子模板就会非常容易。

让我们通过uikit这个CSS框架来完成父模板\_\_base\_\_.html的编写：

**<!DOCTYPE html>**

<html>

<head>

<meta charset="utf-8" />

{% block meta %}*<!-- block meta -->*{% endblock %}

<title>{% block title %} ? {% endblock %} - Awesome Python Webapp</title>

<link rel="stylesheet" href="/static/css/uikit.min.css">

<link rel="stylesheet" href="/static/css/uikit.gradient.min.css">

<link rel="stylesheet" href="/static/css/awesome.css" />

<script src="/static/js/jquery.min.js"></script>

<script src="/static/js/md5.js"></script>

<script src="/static/js/uikit.min.js"></script>

<script src="/static/js/awesome.js"></script>

{% block beforehead %}*<!-- before head -->*{% endblock %}

</head>

<body>

<nav class="uk-navbar uk-navbar-attached uk-margin-bottom">

<div class="uk-container uk-container-center">

<a href="/" class="uk-navbar-brand">Awesome</a>

<ul class="uk-navbar-nav">

<li data-url="blogs"><a href="/"><i class="uk-icon-home"></i> 日志</a></li>

<li><a target="\_blank" href="#"><i class="uk-icon-book"></i> 教程</a></li>

<li><a target="\_blank" href="#"><i class="uk-icon-code"></i> 源码</a></li>

</ul>

<div class="uk-navbar-flip">

<ul class="uk-navbar-nav">

{% if user %}

<li class="uk-parent" data-uk-dropdown>

<a href="#0"><i class="uk-icon-user"></i> {{ user.name }}</a>

<div class="uk-dropdown uk-dropdown-navbar">

<ul class="uk-nav uk-nav-navbar">

<li><a href="/signout"><i class="uk-icon-sign-out"></i> 登出</a></li>

</ul>

</div>

</li>

{% else %}

<li><a href="/signin"><i class="uk-icon-sign-in"></i> 登陆</a></li>

<li><a href="/register"><i class="uk-icon-edit"></i> 注册</a></li>

{% endif %}

</ul>

</div>

</div>

</nav>

<div class="uk-container uk-container-center">

<div class="uk-grid">

*<!-- content -->*

{% block content %}

{% endblock %}

*<!-- // content -->*

</div>

</div>

<div class="uk-margin-large-top" style="background-color:#eee; border-top:1px solid #ccc;">

<div class="uk-container uk-container-center uk-text-center">

<div class="uk-panel uk-margin-top uk-margin-bottom">

<p>

<a target="\_blank" href="#" class="uk-icon-button uk-icon-weibo"></a>

<a target="\_blank" href="#" class="uk-icon-button uk-icon-github"></a>

<a target="\_blank" href="#" class="uk-icon-button uk-icon-linkedin-square"></a>

<a target="\_blank" href="#" class="uk-icon-button uk-icon-twitter"></a>

</p>

<p>Powered by <a href="#">Awesome Python Webapp</a>. Copyright &copy; 2014. [<a href="/manage/" target="\_blank">Manage</a>]</p>

<p><a href="http://www.liaoxuefeng.com/" target="\_blank">www.liaoxuefeng.com</a>. All rights reserved.</p>

<a target="\_blank" href="#"><i class="uk-icon-html5" style="font-size:64px; color: #444;"></i></a>

</div>

</div>

</div>

</body>

</html>

\_\_base\_\_.html定义的几个block作用如下：

用于子页面定义一些meta，例如rss feed：

{% b**lock** meta %} ... {% endblock %}

覆盖页面的标题：

{% b**lock** title %} ... {% endblock %}

子页面可以在<head>标签关闭前插入JavaScript代码：

{% b**lock** beforehead %} ... {% endblock %}

子页面的content布局和内容：

{% b**lock** content %}

...

{% endblock %}

我们把首页改造一下，从\_\_base\_\_.html继承一个blogs.html：

{% extends '\_\_base\_\_.html' %}

{% block title %}日志{% endblock %}

{% block content %}

<div class="uk-width-medium-3-4">

{% for blog in blogs %}

<article class="uk-article">

<h2><a href="/blog/{{ blog.id }}">{{ blog.name }}</a></h2>

<p class="uk-article-meta">发表于{{ blog.created\_at}}</p>

<p>{{ blog.summary }}</p>

<p><a href="/blog/{{ blog.id }}">继续阅读 <i class="uk-icon-angle-double-right"></i></a></p>

</article>

<hr class="uk-article-divider">

{% endfor %}

</div>

<div class="uk-width-medium-1-4">

<div class="uk-panel uk-panel-header">

<h3 class="uk-panel-title">友情链接</h3>

<ul class="uk-list uk-list-line">

<li><i class="uk-icon-thumbs-o-up"></i> <a target="\_blank" href="#">编程</a></li>

<li><i class="uk-icon-thumbs-o-up"></i> <a target="\_blank" href="#">读书</a></li>

<li><i class="uk-icon-thumbs-o-up"></i> <a target="\_blank" href="#">Python教程</a></li>

<li><i class="uk-icon-thumbs-o-up"></i> <a target="\_blank" href="#">Git教程</a></li>

</ul>

</div>

</div>

{% endblock %}

相应地，首页URL的处理函数更新如下：

@get('/')

**def** **index**(request):

summary = 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.'

blogs = [

Blog(id='1', name='Test Blog', summary=summary, created\_at=time.time()-120),

Blog(id='2', name='Something New', summary=summary, created\_at=time.time()-3600),

Blog(id='3', name='Learn Swift', summary=summary, created\_at=time.time()-7200)

]

**return** {

'\_\_template\_\_': 'blogs.html',

'blogs': blogs

}

Blog的创建日期显示的是一个浮点数，因为它是由这段模板渲染出来的：

<p class="uk-article-meta">发表于{{ blog.created\_at }}</p>

解决方法是通过jinja2的filter（过滤器），把一个浮点数转换成日期字符串。我们来编写一个datetime的filter，在模板里用法如下：

<p class="uk-article-meta">发表于{{ blog.created\_at|datetime }}</p>

filter需要在初始化jinja2时设置。相关代码如下：

**def** **datetime\_filter**(t):

delta = int(time.time() - t)

**if** delta < 60:

**return** '1分钟前'

**if** delta < 3600:

**return** '%s分钟前' % (delta // 60)

**if** delta < 86400:

**return** '%s小时前' % (delta // 3600)

**if** delta < 604800:

**return** '%s天前' % (delta // 86400)

dt = datetime.fromtimestamp(t)

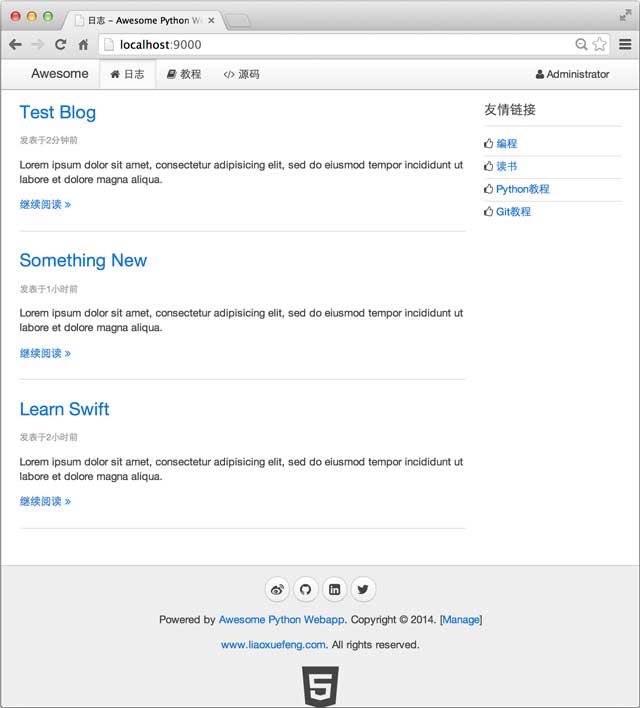
**return** '%s年%s月%s日' % (dt.year, dt.month, dt.day)

...

init\_jinja2(app, filters=dict(datetime=datetime\_filter))

...

现在，完善的首页显示如下：



### 参考源码

[day-08](https://github.com/michaelliao/awesome-python3-webapp/tree/day-08)

### 感觉本站内容不错，读后有收获？

#### Day 9 - 编写API

阅读: 13960

自从Roy Fielding博士在2000年他的博士论文中提出[REST](http://zh.wikipedia.org/wiki/REST)（Representational State Transfer）风格的软件架构模式后，REST就基本上迅速取代了复杂而笨重的SOAP，成为Web API的标准了。

什么是Web API呢？

如果我们想要获取一篇Blog，输入http://localhost:9000/blog/123，就可以看到id为123的Blog页面，但这个结果是HTML页面，它同时混合包含了Blog的数据和Blog的展示两个部分。对于用户来说，阅读起来没有问题，但是，如果机器读取，就很难从HTML中解析出Blog的数据。

如果一个URL返回的不是HTML，而是机器能直接解析的数据，这个URL就可以看成是一个Web API。比如，读取http://localhost:9000/api/blogs/123，如果能直接返回Blog的数据，那么机器就可以直接读取。

REST就是一种设计API的模式。最常用的数据格式是JSON。由于JSON能直接被JavaScript读取，所以，以JSON格式编写的REST风格的API具有简单、易读、易用的特点。

编写API有什么好处呢？由于API就是把Web App的功能全部封装了，所以，通过API操作数据，可以极大地把前端和后端的代码隔离，使得后端代码易于测试，前端代码编写更简单。

一个API也是一个URL的处理函数，我们希望能直接通过一个@api来把函数变成JSON格式的REST API，这样，获取注册用户可以用一个API实现如下：

@get('/api/users')

**def** **api\_get\_users**(\*, page='1'):

page\_index = get\_page\_index(page)

num = **yield** **from** User.findNumber('count(id)')

p = Page(num, page\_index)

**if** num == 0:

**return** dict(page=p, users=())

users = **yield** **from** User.findAll(orderBy='created\_at desc', limit=(p.offset, p.limit))

**for** u **in** users:

u.passwd = '\*\*\*\*\*\*'

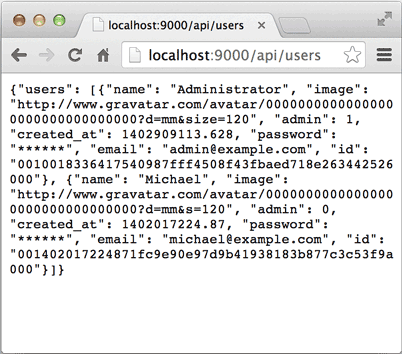
**return** dict(page=p, users=users)

只要返回一个dict，后续的response这个middleware就可以把结果序列化为JSON并返回。

我们需要对Error进行处理，因此定义一个APIError，这种Error是指API调用时发生了逻辑错误（比如用户不存在），其他的Error视为Bug，返回的错误代码为internalerror。

客户端调用API时，必须通过错误代码来区分API调用是否成功。错误代码是用来告诉调用者出错的原因。很多API用一个整数表示错误码，这种方式很难维护错误码，客户端拿到错误码还需要查表得知错误信息。更好的方式是用字符串表示错误代码，不需要看文档也能猜到错误原因。

可以在浏览器直接测试API，例如，输入http://localhost:9000/api/users，就可以看到返回的JSON：



### 参考源码

[day-09](https://github.com/michaelliao/awesome-python3-webapp/tree/day-09)

### 感觉本站内容不错，读后有收获？

#### Day 10 - 用户注册和登录

阅读: 22972

用户管理是绝大部分Web网站都需要解决的问题。用户管理涉及到用户注册和登录。

用户注册相对简单，我们可以先通过API把用户注册这个功能实现了：

\_RE\_EMAIL = re.compile(r'^[a-z0-9\.\-\\_]+\@[a-z0-9\-\\_]+(\.[a-z0-9\-\\_]+){1,4}$')

\_RE\_SHA1 = re.compile(r'^[0-9a-f]{40}$')

@post('/api/users')

**def** **api\_register\_user**(\*, email, name, passwd):

**if** **not** name **or** **not** name.strip():

**raise** APIValueError('name')

**if** **not** email **or** **not** \_RE\_EMAIL.match(email):

**raise** APIValueError('email')

**if** **not** passwd **or** **not** \_RE\_SHA1.match(passwd):

**raise** APIValueError('passwd')

users = **yield** **from** User.findAll('email=?', [email])

**if** len(users) > 0:

**raise** APIError('register:failed', 'email', 'Email is already in use.')

uid = next\_id()

sha1\_passwd = '%s:%s' % (uid, passwd)

user = User(id=uid, name=name.strip(), email=email, passwd=hashlib.sha1(sha1\_passwd.encode('utf-8')).hexdigest(), image='http://www.gravatar.com/avatar/%s?d=mm&s=120' % hashlib.md5(email.encode('utf-8')).hexdigest())

**yield** **from** user.save()

*# make session cookie:*

r = web.Response()

r.set\_cookie(COOKIE\_NAME, user2cookie(user, 86400), max\_age=86400, httponly=True)

user.passwd = '\*\*\*\*\*\*'

r.content\_type = 'application/json'

r.body = json.dumps(user, ensure\_ascii=False).encode('utf-8')

**return** r

注意用户口令是客户端传递的经过SHA1计算后的40位Hash字符串，所以服务器端并不知道用户的原始口令。

接下来可以创建一个注册页面，让用户填写注册表单，然后，提交数据到注册用户的API：

{% extends '\_\_base\_\_.html' %}

{% block title %}注册{% endblock %}

{% block beforehead %}

<script>

**function** validateEmail(email) {

**var** re = /^[a-z0-9\.\-\\_]+\@[a-z0-9\-\\_]+(\.[a-z0-9\-\\_]+){1,4}$/;

**return** re.test(email.toLowerCase());

}

$(**function** () {

**var** vm = **new** Vue({

el: '#vm',

data: {

name: '',

email: '',

password1: '',

password2: ''

},

methods: {

submit: **function** (event) {

event.preventDefault();

**var** $form = $('#vm');

**if** (! **this**.name.trim()) {

**return** $form.showFormError('请输入名字');

}

**if** (! validateEmail(**this**.email.trim().toLowerCase())) {

**return** $form.showFormError('请输入正确的Email地址');

}

**if** (**this**.password1.length < 6) {

**return** $form.showFormError('口令长度至少为6个字符');

}

**if** (**this**.password1 !== **this**.password2) {

**return** $form.showFormError('两次输入的口令不一致');

}

**var** email = **this**.email.trim().toLowerCase();

$form.postJSON('/api/users', {

name: **this**.name.trim(),

email: email,

passwd: CryptoJS.SHA1(email + ':' + **this**.password1).toString()

}, **function** (err, r) {

**if** (err) {

**return** $form.showFormError(err);

}

**return** location.assign('/');

});

}

}

});

$('#vm').show();

});

</script>

{% endblock %}

{% block content %}

<div class="uk-width-2-3">

<h1>欢迎注册！</h1>

<form id="vm" v-on="submit: submit" class="uk-form uk-form-stacked">

<div class="uk-alert uk-alert-danger uk-hidden"></div>

<div class="uk-form-row">

<label class="uk-form-label">名字:</label>

<div class="uk-form-controls">

<input v-model="name" type="text" maxlength="50" placeholder="名字" class="uk-width-1-1">

</div>

</div>

<div class="uk-form-row">

<label class="uk-form-label">电子邮件:</label>

<div class="uk-form-controls">

<input v-model="email" type="text" maxlength="50" placeholder="your-name@example.com" class="uk-width-1-1">

</div>

</div>

<div class="uk-form-row">

<label class="uk-form-label">输入口令:</label>

<div class="uk-form-controls">

<input v-model="password1" type="password" maxlength="50" placeholder="输入口令" class="uk-width-1-1">

</div>

</div>

<div class="uk-form-row">

<label class="uk-form-label">重复口令:</label>

<div class="uk-form-controls">

<input v-model="password2" type="password" maxlength="50" placeholder="重复口令" class="uk-width-1-1">

</div>

</div>

<div class="uk-form-row">

<button type="submit" class="uk-button uk-button-primary"><i class="uk-icon-user"></i> 注册</button>

</div>

</form>

</div>

{% endblock %}

这样我们就把用户注册的功能完成了：



用户登录比用户注册复杂。由于HTTP协议是一种无状态协议，而服务器要跟踪用户状态，就只能通过cookie实现。大多数Web框架提供了Session功能来封装保存用户状态的cookie。

Session的优点是简单易用，可以直接从Session中取出用户登录信息。

Session的缺点是服务器需要在内存中维护一个映射表来存储用户登录信息，如果有两台以上服务器，就需要对Session做集群，因此，使用Session的Web App很难扩展。

我们采用直接读取cookie的方式来验证用户登录，每次用户访问任意URL，都会对cookie进行验证，这种方式的好处是保证服务器处理任意的URL都是无状态的，可以扩展到多台服务器。

由于登录成功后是由服务器生成一个cookie发送给浏览器，所以，要保证这个cookie不会被客户端伪造出来。

实现防伪造cookie的关键是通过一个单向算法（例如SHA1），举例如下：

当用户输入了正确的口令登录成功后，服务器可以从数据库取到用户的id，并按照如下方式计算出一个字符串：

"用户id" + "过期时间" + SHA1("用户id" + "用户口令" + "过期时间" + "SecretKey")

当浏览器发送cookie到服务器端后，服务器可以拿到的信息包括：

* 用户id
* 过期时间
* SHA1值

如果未到过期时间，服务器就根据用户id查找用户口令，并计算：

SHA1("用户id" + "用户口令" + "过期时间" + "SecretKey")

并与浏览器cookie中的MD5进行比较，如果相等，则说明用户已登录，否则，cookie就是伪造的。

这个算法的关键在于SHA1是一种单向算法，即可以通过原始字符串计算出SHA1结果，但无法通过SHA1结果反推出原始字符串。

所以登录API可以实现如下：

@post('/api/authenticate')

**def** **authenticate**(\*, email, passwd):

**if** **not** email:

**raise** APIValueError('email', 'Invalid email.')

**if** **not** passwd:

**raise** APIValueError('passwd', 'Invalid password.')

users = **yield** **from** User.findAll('email=?', [email])

**if** len(users) == 0:

**raise** APIValueError('email', 'Email not exist.')

user = users[0]

*# check passwd:*

sha1 = hashlib.sha1()

sha1.update(user.id.encode('utf-8'))

sha1.update(b':')

sha1.update(passwd.encode('utf-8'))

**if** user.passwd != sha1.hexdigest():

**raise** APIValueError('passwd', 'Invalid password.')

*# authenticate ok, set cookie:*

r = web.Response()

r.set\_cookie(COOKIE\_NAME, user2cookie(user, 86400), max\_age=86400, httponly=True)

user.passwd = '\*\*\*\*\*\*'

r.content\_type = 'application/json'

r.body = json.dumps(user, ensure\_ascii=False).encode('utf-8')

**return** r

*# 计算加密cookie:*

**def** **user2cookie**(user, max\_age):

*# build cookie string by: id-expires-sha1*

expires = str(int(time.time() + max\_age))

s = '%s-%s-%s-%s' % (user.id, user.passwd, expires, \_COOKIE\_KEY)

L = [user.id, expires, hashlib.sha1(s.encode('utf-8')).hexdigest()]

**return** '-'.join(L)

对于每个URL处理函数，如果我们都去写解析cookie的代码，那会导致代码重复很多次。

利用middle在处理URL之前，把cookie解析出来，并将登录用户绑定到request对象上，这样，后续的URL处理函数就可以直接拿到登录用户：

@asyncio.coroutine

**def** **auth\_factory**(app, handler):

@asyncio.coroutine

**def** **auth**(request):

logging.info('check user: %s %s' % (request.method, request.path))

request.\_\_user\_\_ = None

cookie\_str = request.cookies.get(COOKIE\_NAME)

**if** cookie\_str:

user = **yield** **from** cookie2user(cookie\_str)

**if** user:

logging.info('set current user: %s' % user.email)

request.\_\_user\_\_ = user

**return** (**yield** **from** handler(request))

**return** auth

*# 解密cookie:*

@asyncio.coroutine

**def** **cookie2user**(cookie\_str):

'''

Parse cookie and load user if cookie is valid.

'''

**if** **not** cookie\_str:

**return** None

**try**:

L = cookie\_str.split('-')

**if** len(L) != 3:

**return** None

uid, expires, sha1 = L

**if** int(expires) < time.time():

**return** None

user = **yield** **from** User.find(uid)

**if** user **is** None:

**return** None

s = '%s-%s-%s-%s' % (uid, user.passwd, expires, \_COOKIE\_KEY)

**if** sha1 != hashlib.sha1(s.encode('utf-8')).hexdigest():

logging.info('invalid sha1')

**return** None

user.passwd = '\*\*\*\*\*\*'

**return** user

**except** Exception **as** e:

logging.exception(e)

**return** None

这样，我们就完成了用户注册和登录的功能。

### 参考源码

[day-10](https://github.com/michaelliao/awesome-python3-webapp/tree/day-10)

### 感觉本站内容不错，读后有收获？

#### Day 11 - 编写日志创建页

阅读: 547

在Web开发中，后端代码写起来其实是相当容易的。

例如，我们编写一个REST API，用于创建一个Blog：

@post('/api/blogs')

**def** **api\_create\_blog**(request, \*, name, summary, content):

check\_admin(request)

**if** **not** name **or** **not** name.strip():

**raise** APIValueError('name', 'name cannot be empty.')

**if** **not** summary **or** **not** summary.strip():

**raise** APIValueError('summary', 'summary cannot be empty.')

**if** **not** content **or** **not** content.strip():

**raise** APIValueError('content', 'content cannot be empty.')

blog = Blog(user\_id=request.\_\_user\_\_.id, user\_name=request.\_\_user\_\_.name, user\_image=request.\_\_user\_\_.image, name=name.strip(), summary=summary.strip(), content=content.strip())

**yield** **from** blog.save()

**return** blog

编写后端Python代码不但很简单，而且非常容易测试，上面的API：api\_create\_blog()本身只是一个普通函数。

Web开发真正困难的地方在于编写前端页面。前端页面需要混合HTML、CSS和JavaScript，如果对这三者没有深入地掌握，编写的前端页面将很快难以维护。

更大的问题在于，前端页面通常是动态页面，也就是说，前端页面往往是由后端代码生成的。

生成前端页面最早的方式是拼接字符串：

s = '<html><head><title>'

+ title

+ '</title></head><body>'

+ body

+ '</body></html>'

显然这种方式完全不具备可维护性。所以有第二种模板方式：

<html>

<head>

<title>{{ title }}</title>

</head>

<body>

{{ body }}

</body>

</html>

ASP、JSP、PHP等都是用这种模板方式生成前端页面。

如果在页面上大量使用JavaScript（事实上大部分页面都会），模板方式仍然会导致JavaScript代码与后端代码绑得非常紧密，以至于难以维护。其根本原因在于负责显示的HTML DOM模型与负责数据和交互的JavaScript代码没有分割清楚。

要编写可维护的前端代码绝非易事。和后端结合的MVC模式已经无法满足复杂页面逻辑的需要了，所以，新的[MVVM](http://en.wikipedia.org/wiki/Model_View_ViewModel)：Model View ViewModel模式应运而生。

MVVM最早由微软提出来，它借鉴了桌面应用程序的MVC思想，在前端页面中，把Model用纯JavaScript对象表示：

<script>

**var** blog = {

name: 'hello',

summary: 'this is summary',

content: 'this is content...'

};

</script>

View是纯HTML：

<form action="/api/blogs" method="post">

<input name="name">

<input name="summary">

<textarea name="content"></textarea>

<button type="submit">OK</button>

</form>

由于Model表示数据，View负责显示，两者做到了最大限度的分离。

把Model和View关联起来的就是ViewModel。ViewModel负责把Model的数据同步到View显示出来，还负责把View的修改同步回Model。

ViewModel如何编写？需要用JavaScript编写一个通用的ViewModel，这样，就可以复用整个MVVM模型了。

好消息是已有许多成熟的MVVM框架，例如AngularJS，KnockoutJS等。我们选择[Vue](http://vuejs.org/)这个简单易用的MVVM框架来实现创建Blog的页面templates/manage\_blog\_edit.html：

{% extends '\_\_base\_\_.html' %}

{% block title %}编辑日志{% endblock %}

{% block beforehead %}

<script>

**var**

ID = '{{ id }}',

action = '{{ action }}';

**function** initVM(blog) {

**var** vm = **new** Vue({

el: '#vm',

data: blog,

methods: {

submit: **function** (event) {

event.preventDefault();

**var** $form = $('#vm').find('form');

$form.postJSON(action, **this**.$data, **function** (err, r) {

**if** (err) {

$form.showFormError(err);

}

**else** {

**return** location.assign('/api/blogs/' + r.id);

}

});

}

}

});

$('#vm').show();

}

$(**function** () {

**if** (ID) {

getJSON('/api/blogs/' + ID, **function** (err, blog) {

**if** (err) {

**return** fatal(err);

}

$('#loading').hide();

initVM(blog);

});

}

**else** {

$('#loading').hide();

initVM({

name: '',

summary: '',

content: ''

});

}

});

</script>

{% endblock %}

{% block content %}

<div class="uk-width-1-1 uk-margin-bottom">

<div class="uk-panel uk-panel-box">

<ul class="uk-breadcrumb">

<li><a href="/manage/comments">评论</a></li>

<li><a href="/manage/blogs">日志</a></li>

<li><a href="/manage/users">用户</a></li>

</ul>

</div>

</div>

<div id="error" class="uk-width-1-1">

</div>

<div id="loading" class="uk-width-1-1 uk-text-center">

<span><i class="uk-icon-spinner uk-icon-medium uk-icon-spin"></i> 正在加载...</span>

</div>

<div id="vm" class="uk-width-2-3">

<form v-on="submit: submit" class="uk-form uk-form-stacked">

<div class="uk-alert uk-alert-danger uk-hidden"></div>

<div class="uk-form-row">

<label class="uk-form-label">标题:</label>

<div class="uk-form-controls">

<input v-model="name" name="name" type="text" placeholder="标题" class="uk-width-1-1">

</div>

</div>

<div class="uk-form-row">

<label class="uk-form-label">摘要:</label>

<div class="uk-form-controls">

<textarea v-model="summary" rows="4" name="summary" placeholder="摘要" class="uk-width-1-1" style="resize:none;"></textarea>

</div>

</div>

<div class="uk-form-row">

<label class="uk-form-label">内容:</label>

<div class="uk-form-controls">

<textarea v-model="content" rows="16" name="content" placeholder="内容" class="uk-width-1-1" style="resize:none;"></textarea>

</div>

</div>

<div class="uk-form-row">

<button type="submit" class="uk-button uk-button-primary"><i class="uk-icon-save"></i> 保存</button>

<a href="/manage/blogs" class="uk-button"><i class="uk-icon-times"></i> 取消</a>

</div>

</form>

</div>

{% endblock %}

初始化Vue时，我们指定3个参数：

el：根据选择器查找绑定的View，这里是#vm，就是id为vm的DOM，对应的是一个<div>标签；

data：JavaScript对象表示的Model，我们初始化为{ name: '', summary: '', content: ''}；

methods：View可以触发的JavaScript函数，submit就是提交表单时触发的函数。

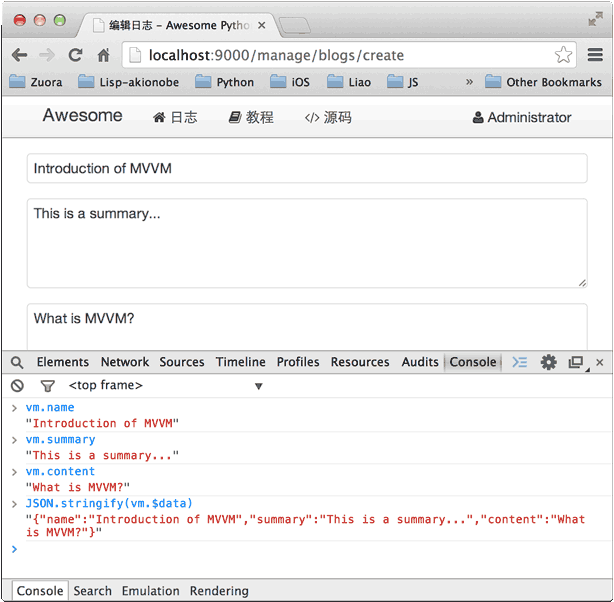
接下来，我们在<form>标签中，用几个简单的v-model，就可以让Vue把Model和View关联起来：

*<!-- input的value和Model的name关联起来了 -->*

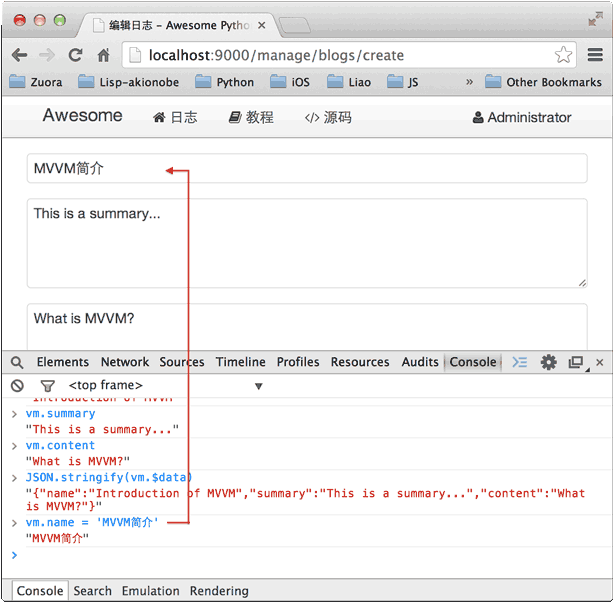
<input v-model="name" class="uk-width-1-1">

Form表单通过<form v-on="submit: submit">把提交表单的事件关联到submit方法。

需要特别注意的是，在MVVM中，Model和View是双向绑定的。如果我们在Form中修改了文本框的值，可以在Model中立刻拿到新的值。试试在表单中输入文本，然后在Chrome浏览器中打开JavaScript控制台，可以通过vm.name访问单个属性，或者通过vm.$data访问整个Model：



如果我们在JavaScript逻辑中修改了Model，这个修改会立刻反映到View上。试试在JavaScript控制台输入vm.name = 'MVVM简介'，可以看到文本框的内容自动被同步了：



双向绑定是MVVM框架最大的作用。借助于MVVM，我们把复杂的显示逻辑交给框架完成。由于后端编写了独立的REST API，所以，前端用AJAX提交表单非常容易，前后端分离得非常彻底。

### 参考源码

[day-11](https://github.com/michaelliao/awesome-python3-webapp/tree/day-11)

### 感觉本站内容不错，读后有收获？

#### Day 12 - 编写日志列表页

阅读: 231

MVVM模式不但可用于Form表单，在复杂的管理页面中也能大显身手。例如，分页显示Blog的功能，我们先把后端代码写出来：

在apis.py中定义一个Page类用于存储分页信息：

**class Page(object):**

**def** **\_\_init\_\_**(self, item\_count, page\_index=1, page\_size=10):

self.item\_count = item\_count

self.page\_size = page\_size

self.page\_count = item\_count // page\_size + (1 **if** item\_count % page\_size > 0 **else** 0)

**if** (item\_count == 0) **or** (page\_index > self.page\_count):

self.offset = 0

self.limit = 0

self.page\_index = 1

**else**:

self.page\_index = page\_index

self.offset = self.page\_size \* (page\_index - 1)

self.limit = self.page\_size

self.has\_next = self.page\_index < self.page\_count

self.has\_previous = self.page\_index > 1

**def** **\_\_str\_\_**(self):

**return** 'item\_count: %s, page\_count: %s, page\_index: %s, page\_size: %s, offset: %s, limit: %s' % (self.item\_count, self.page\_count, self.page\_index, self.page\_size, self.offset, self.limit)

\_\_repr\_\_ = \_\_str\_\_

在handlers.py中实现API：

@get('/api/blogs')

**def** **api\_blogs**(\*, page='1'):

page\_index = get\_page\_index(page)

num = **yield** **from** Blog.findNumber('count(id)')

p = Page(num, page\_index)

**if** num == 0:

**return** dict(page=p, blogs=())

blogs = **yield** **from** Blog.findAll(orderBy='created\_at desc', limit=(p.offset, p.limit))

**return** dict(page=p, blogs=blogs)

管理页面：

@get('/manage/blogs')

**def** **manage\_blogs**(\*, page='1'):

**return** {

'\_\_template\_\_': 'manage\_blogs.html',

'page\_index': get\_page\_index(page)

}

模板页面首先通过API：GET /api/blogs?page=?拿到Model：

{

"page": {

"has\_next": true,

"page\_index": 1,

"page\_count": 2,

"has\_previous": false,

"item\_count": 12

},

"blogs": [...]

}

然后，通过Vue初始化MVVM：

<script>

**function** initVM(data) {

**var** vm = **new** Vue({

el: '#vm',

data: {

blogs: data.blogs,

page: data.page

},

methods: {

edit\_blog: **function** (blog) {

location.assign('/manage/blogs/edit?id=' + blog.id);

},

delete\_blog: **function** (blog) {

**if** (confirm('确认要删除“' + blog.name + '”？删除后不可恢复！')) {

postJSON('/api/blogs/' + blog.id + '/delete', **function** (err, r) {

**if** (err) {

**return** alert(err.message || err.error || err);

}

refresh();

});

}

}

}

});

$('#vm').show();

}

$(**function**() {

getJSON('/api/blogs', {

page: {{ page\_index }}

}, **function** (err, results) {

**if** (err) {

**return** fatal(err);

}

$('#loading').hide();

initVM(results);

});

});

</script>

View的容器是#vm，包含一个table，我们用v-repeat可以把Model的数组blogs直接变成多行的<tr>：

<div id="vm" class="uk-width-1-1">

<a href="/manage/blogs/create" class="uk-button uk-button-primary"><i class="uk-icon-plus"></i> 新日志</a>

<table class="uk-table uk-table-hover">

<thead>

<tr>

<th class="uk-width-5-10">标题 / 摘要</th>

<th class="uk-width-2-10">作者</th>

<th class="uk-width-2-10">创建时间</th>

<th class="uk-width-1-10">操作</th>

</tr>

</thead>

<tbody>

<tr v-repeat="blog: blogs" >

<td>

<a target="\_blank" v-attr="href: '/blog/'+blog.id" v-text="blog.name"></a>

</td>

<td>

<a target="\_blank" v-attr="href: '/user/'+blog.user\_id" v-text="blog.user\_name"></a>

</td>

<td>

<span v-text="blog.created\_at.toDateTime()"></span>

</td>

<td>

<a href="#0" v-on="click: edit\_blog(blog)"><i class="uk-icon-edit"></i>

<a href="#0" v-on="click: delete\_blog(blog)"><i class="uk-icon-trash-o"></i>

</td>

</tr>

</tbody>

</table>

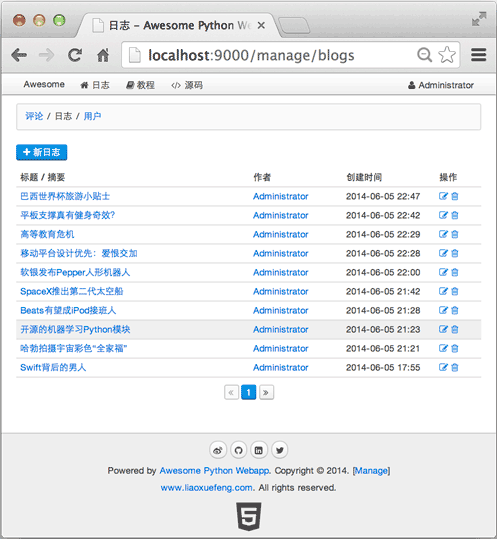
<div v-component="pagination" v-with="page"></div>

</div>

往Model的blogs数组中增加一个Blog元素，table就神奇地增加了一行；把blogs数组的某个元素删除，table就神奇地减少了一行。所有复杂的Model-View的映射逻辑全部由MVVM框架完成，我们只需要在HTML中写上v-repeat指令，就什么都不用管了。

可以把v-repeat="blog: blogs"看成循环代码，所以，可以在一个<tr>内部引用循环变量blog。v-text和v-attr指令分别用于生成文本和DOM节点属性。

完整的Blog列表页如下：



### 参考源码

[day-12](https://github.com/michaelliao/awesome-python3-webapp/tree/day-12)

### 感觉本站内容不错，读后有收获？

#### Day 13 - 提升开发效率

阅读: 1250

现在，我们已经把一个Web App的框架完全搭建好了，从后端的API到前端的MVVM，流程已经跑通了。

在继续工作前，注意到每次修改Python代码，都必须在命令行先Ctrl-C停止服务器，再重启，改动才能生效。

在开发阶段，每天都要修改、保存几十次代码，每次保存都手动来这么一下非常麻烦，严重地降低了我们的开发效率。有没有办法让服务器检测到代码修改后自动重新加载呢？

Django的开发环境在Debug模式下就可以做到自动重新加载，如果我们编写的服务器也能实现这个功能，就能大大提升开发效率。

可惜的是，Django没把这个功能独立出来，不用Django就享受不到，怎么办？

其实Python本身提供了重新载入模块的功能，但不是所有模块都能被重新载入。另一种思路是检测www目录下的代码改动，一旦有改动，就自动重启服务器。

按照这个思路，我们可以编写一个辅助程序pymonitor.py，让它启动wsgiapp.py，并时刻监控www目录下的代码改动，有改动时，先把当前wsgiapp.py进程杀掉，再重启，就完成了服务器进程的自动重启。

要监控目录文件的变化，我们也无需自己手动定时扫描，Python的第三方库watchdog可以利用操作系统的API来监控目录文件的变化，并发送通知。我们先用pip安装：

$ pip3 install watchdog

利用watchdog接收文件变化的通知，如果是.py文件，就自动重启wsgiapp.py进程。

利用Python自带的subprocess实现进程的启动和终止，并把输入输出重定向到当前进程的输入输出中：

*#!/usr/bin/env python3*

*# -\*- coding: utf-8 -\*-*

\_\_author\_\_ = 'Michael Liao'

**import** os, sys, time, subprocess

**from** watchdog.observers **import** Observer

**from** watchdog.events **import** FileSystemEventHandler

**def** **log**(s):

print('[Monitor] %s' % s)

**class MyFileSystemEventHander(FileSystemEventHandler):**

**def** **\_\_init\_\_**(self, fn):

super(MyFileSystemEventHander, self).\_\_init\_\_()

self.restart = fn

**def** **on\_any\_event**(self, event):

**if** event.src\_path.endswith('.py'):

log('Python source file changed: %s' % event.src\_path)

self.restart()

command = ['echo', 'ok']

process = None

**def** **kill\_process**():

**global** process

**if** process:

log('Kill process [%s]...' % process.pid)

process.kill()

process.wait()

log('Process ended with code %s.' % process.returncode)

process = None

**def** **start\_process**():

**global** process, command

log('Start process %s...' % ' '.join(command))

process = subprocess.Popen(command, stdin=sys.stdin, stdout=sys.stdout, stderr=sys.stderr)

**def** **restart\_process**():

kill\_process()

start\_process()

**def** **start\_watch**(path, callback):

observer = Observer()

observer.schedule(MyFileSystemEventHander(restart\_process), path, recursive=True)

observer.start()

log('Watching directory %s...' % path)

start\_process()

**try**:

**while** True:

time.sleep(0.5)

**except** KeyboardInterrupt:

observer.stop()

observer.join()

**if** \_\_name\_\_ == '\_\_main\_\_':

argv = sys.argv[1:]

**if** **not** argv:

print('Usage: ./pymonitor your-script.py')

exit(0)

**if** argv[0] != 'python3':

argv.insert(0, 'python3')

command = argv

path = os.path.abspath('.')

start\_watch(path, None)

一共70行左右的代码，就实现了Debug模式的自动重新加载。用下面的命令启动服务器：

$ python3 pymonitor.py wsgiapp.py

或者给pymonitor.py加上可执行权限，启动服务器：

$ ./pymonitor.py app.py

在编辑器中打开一个.py文件，修改后保存，看看命令行输出，是不是自动重启了服务器：

$ ./pymonitor.py app.py

[Monitor] Watching directory /Users/michael/Github/awesome-python3-webapp/www...

[Monitor] **Start** process python app.py...

...

INFO:root:application (/Users/michael/Github/awesome-python3-webapp/www) will **start** **at** 0.0.0.0:9000...

[Monitor] Python source file changed: /Users/michael/Github/awesome-python-webapp/www/handlers.py

[Monitor] Kill process [2747]...

[Monitor] Process ended **with** code -9.

[Monitor] **Start** process python app.py...

...

INFO:root:application (/Users/michael/Github/awesome-python3-webapp/www) will **start** **at** 0.0.0.0:9000...

现在，只要一保存代码，就可以刷新浏览器看到效果，大大提升了开发效率。

### 感觉本站内容不错，读后有收获？

#### Day 14 - 完成Web App

阅读: 21559

在Web App框架和基本流程跑通后，剩下的工作全部是体力活了：在Debug开发模式下完成后端所有API、前端所有页面。我们需要做的事情包括：

把当前用户绑定到request上，并对URL/manage/进行拦截，检查当前用户是否是管理员身份：

@asyncio.coroutine

**def** **auth\_factory**(app, handler):

@asyncio.coroutine

**def** **auth**(request):

logging.info('check user: %s %s' % (request.method, request.path))

request.\_\_user\_\_ = None

cookie\_str = request.cookies.get(COOKIE\_NAME)

**if** cookie\_str:

user = **yield** **from** cookie2user(cookie\_str)

**if** user:

logging.info('set current user: %s' % user.email)

request.\_\_user\_\_ = user

**if** request.path.startswith('/manage/') **and** (request.\_\_user\_\_ **is** None **or** **not** request.\_\_user\_\_.admin):

**return** web.HTTPFound('/signin')

**return** (**yield** **from** handler(request))

**return** auth

后端API包括：

* 获取日志：GET /api/blogs
* 创建日志：POST /api/blogs
* 修改日志：POST /api/blogs/:blog\_id
* 删除日志：POST /api/blogs/:blog\_id/delete
* 获取评论：GET /api/comments
* 创建评论：POST /api/blogs/:blog\_id/comments
* 删除评论：POST /api/comments/:comment\_id/delete
* 创建新用户：POST /api/users
* 获取用户：GET /api/users

管理页面包括：

* 评论列表页：GET /manage/comments
* 日志列表页：GET /manage/blogs
* 创建日志页：GET /manage/blogs/create
* 修改日志页：GET /manage/blogs/
* 用户列表页：GET /manage/users

用户浏览页面包括：

* 注册页：GET /register
* 登录页：GET /signin
* 注销页：GET /signout
* 首页：GET /
* 日志详情页：GET /blog/:blog\_id

把所有的功能实现，我们第一个Web App就宣告完成！

### 参考源码

[day-14](https://github.com/michaelliao/awesome-python3-webapp/tree/day-14)

### 感觉本站内容不错，读后有收获？

#### Day 15 - 部署Web App

阅读: 26342

作为一个合格的开发者，在本地环境下完成开发还远远不够，我们需要把Web App部署到远程服务器上，这样，广大用户才能访问到网站。

很多做开发的同学把部署这件事情看成是运维同学的工作，这种看法是完全错误的。首先，最近流行[DevOps](http://zh.wikipedia.org/wiki/DevOps)理念，就是说，开发和运维要变成一个整体。其次，运维的难度，其实跟开发质量有很大的关系。代码写得垃圾，运维再好也架不住天天挂掉。最后，DevOps理念需要把运维、监控等功能融入到开发中。你想服务器升级时不中断用户服务？那就得在开发时考虑到这一点。

下面，我们就来把awesome-python3-webapp部署到Linux服务器。

### 搭建Linux服务器

要部署到Linux，首先得有一台Linux服务器。要在公网上体验的同学，可以在Amazon的[AWS](http://aws.amazon.com/)申请一台EC2虚拟机（免费使用1年），或者使用国内的一些云服务器，一般都提供Ubuntu Server的镜像。想在本地部署的同学，请安装虚拟机，推荐使用[VirtualBox](https://www.virtualbox.org/)。

我们选择的Linux服务器版本是[Ubuntu Server 14.04 LTS](http://www.ubuntu.com/download/server)，原因是apt太简单了。如果你准备使用其他Linux版本，也没有问题。

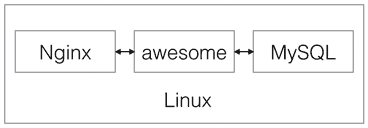
Linux安装完成后，请确保ssh服务正在运行，否则，需要通过apt安装：

$ su**do** apt-**get** install openssh-server

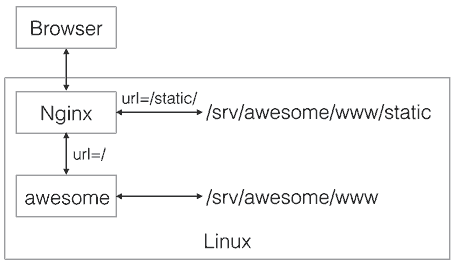
有了ssh服务，就可以从本地连接到服务器上。建议把公钥复制到服务器端用户的.ssh/authorized\_keys中，这样，就可以通过证书实现无密码连接。

### 部署方式

利用Python自带的asyncio，我们已经编写了一个异步高性能服务器。但是，我们还需要一个高性能的Web服务器，这里选择Nginx，它可以处理静态资源，同时作为反向代理把动态请求交给Python代码处理。这个模型如下：



Nginx负责分发请求：



在服务器端，我们需要定义好部署的目录结构：

/

+- srv/

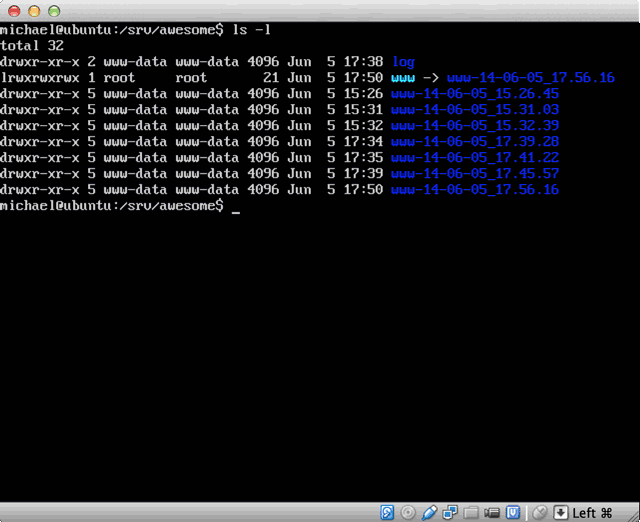
+- awesome/ <-- Web App根目录

+- www/ <-- 存放Python源码

| +- static/ <-- 存放静态资源文件

+- log/ <-- 存放log

在服务器上部署，要考虑到新版本如果运行不正常，需要回退到旧版本时怎么办。每次用新的代码覆盖掉旧的文件是不行的，需要一个类似版本控制的机制。由于Linux系统提供了软链接功能，所以，我们把www作为一个软链接，它指向哪个目录，哪个目录就是当前运行的版本：



而Nginx和python代码的配置文件只需要指向www目录即可。

Nginx可以作为服务进程直接启动，但app.py还不行，所以，[Supervisor](http://supervisord.org/)登场！Supervisor是一个管理进程的工具，可以随系统启动而启动服务，它还时刻监控服务进程，如果服务进程意外退出，Supervisor可以自动重启服务。

总结一下我们需要用到的服务有：

* Nginx：高性能Web服务器+负责反向代理；
* Supervisor：监控服务进程的工具；
* MySQL：数据库服务。

在Linux服务器上用apt可以直接安装上述服务：

$ su**do** apt-**get** install nginx supervisor python3 mysql-server

然后，再把我们自己的Web App用到的Python库安装了：

$ su**do** pip3 install jinja2 aiomysql aiohttp

在服务器上创建目录/srv/awesome/以及相应的子目录。

在服务器上初始化MySQL数据库，把数据库初始化脚本schema.sql复制到服务器上执行：

$ mysql -u root -p < schema.sql

服务器端准备就绪。

### 部署

用FTP还是SCP还是rsync复制文件？如果你需要手动复制，用一次两次还行，一天如果部署50次不但慢、效率低，而且容易出错。

正确的部署方式是使用工具配合脚本完成自动化部署。[Fabric](http://www.fabfile.org/)就是一个自动化部署工具。由于Fabric是用Python 2.x开发的，所以，部署脚本要用Python 2.7来编写，本机还必须安装Python 2.7版本。

要用Fabric部署，需要在本机（是开发机器，不是Linux服务器）安装Fabric：

$ easy\_install fabric

Linux服务器上不需要安装Fabric，Fabric使用SSH直接登录服务器并执行部署命令。

下一步是编写部署脚本。Fabric的部署脚本叫fabfile.py，我们把它放到awesome-python-webapp的目录下，与www目录平级：

awesome-python-webapp/

+- fabfile.py

+- www/

+- ...

Fabric的脚本编写很简单，首先导入Fabric的API，设置部署时的变量：

*# fabfile.py*

**import** os, re

**from** datetime **import** datetime

*# 导入Fabric API:*

**from** fabric.api **import** \*

*# 服务器登录用户名:*

env.user = 'michael'

*# sudo用户为root:*

env.sudo\_user = 'root'

*# 服务器地址，可以有多个，依次部署:*

env.hosts = ['192.168.0.3']

*# 服务器MySQL用户名和口令:*

db\_user = 'www-data'

db\_password = 'www-data'

然后，每个Python函数都是一个任务。我们先编写一个打包的任务：

\_TAR\_FILE = 'dist-awesome.tar.gz'

**def** **build**():

includes = ['static', 'templates', 'transwarp', 'favicon.ico', '\*.py']

excludes = ['test', '.\*', '\*.pyc', '\*.pyo']

local('rm -f dist/%s' % \_TAR\_FILE)

**with** lcd(os.path.join(os.path.abspath('.'), 'www')):

cmd = ['tar', '--dereference', '-czvf', '../dist/%s' % \_TAR\_FILE]

cmd.extend(['--exclude=\'%s\'' % ex **for** ex **in** excludes])

cmd.extend(includes)

local(' '.join(cmd))

Fabric提供local('...')来运行本地命令，with lcd(path)可以把当前命令的目录设定为lcd()指定的目录，注意Fabric只能运行命令行命令，Windows下可能需要[Cgywin](http://cygwin.com/)环境。

在awesome-python-webapp目录下运行：

$ fab build

看看是否在dist目录下创建了dist-awesome.tar.gz的文件。

打包后，我们就可以继续编写deploy任务，把打包文件上传至服务器，解压，重置www软链接，重启相关服务：

\_REMOTE\_TMP\_TAR = '/tmp/%s' % \_TAR\_FILE

\_REMOTE\_BASE\_DIR = '/srv/awesome'

**def** **deploy**():

newdir = 'www-%s' % datetime.now().strftime('%y-%m-%d\_%H.%M.%S')

*# 删除已有的tar文件:*

run('rm -f %s' % \_REMOTE\_TMP\_TAR)

*# 上传新的tar文件:*

put('dist/%s' % \_TAR\_FILE, \_REMOTE\_TMP\_TAR)

*# 创建新目录:*

**with** cd(\_REMOTE\_BASE\_DIR):

sudo('mkdir %s' % newdir)

*# 解压到新目录:*

**with** cd('%s/%s' % (\_REMOTE\_BASE\_DIR, newdir)):

sudo('tar -xzvf %s' % \_REMOTE\_TMP\_TAR)

*# 重置软链接:*

**with** cd(\_REMOTE\_BASE\_DIR):

sudo('rm -f www')

sudo('ln -s %s www' % newdir)

sudo('chown www-data:www-data www')

sudo('chown -R www-data:www-data %s' % newdir)

*# 重启Python服务和nginx服务器:*

**with** settings(warn\_only=True):

sudo('supervisorctl stop awesome')

sudo('supervisorctl start awesome')

sudo('/etc/init.d/nginx reload')

注意run()函数执行的命令是在服务器上运行，with cd(path)和with lcd(path)类似，把当前目录在服务器端设置为cd()指定的目录。如果一个命令需要sudo权限，就不能用run()，而是用sudo()来执行。

### 配置Supervisor

上面让Supervisor重启awesome的命令会失败，因为我们还没有配置Supervisor呢。

编写一个Supervisor的配置文件awesome.conf，存放到/etc/supervisor/conf.d/目录下：

**[program:awesome]**

command = /srv/awesome/www/app.py

directory = /srv/awesome/www

user = www-data

startsecs = 3

redirect\_stderr = **true**

stdout\_logfile\_maxbytes = 50MB

stdout\_logfile\_backups = 10

stdout\_logfile = /srv/awesome/log/app.log

配置文件通过[program:awesome]指定服务名为awesome，command指定启动app.py。

然后重启Supervisor后，就可以随时启动和停止Supervisor管理的服务了：

$ sudo supervisorctl reload

$ sudo supervisorctl start awesome

$ sudo supervisorctl status

awesome RUNNING pid 1401, uptime 5:01:34

### 配置Nginx

Supervisor只负责运行app.py，我们还需要配置Nginx。把配置文件awesome放到/etc/nginx/sites-available/目录下：

server {

listen 80; *# 监听80端口*

root /srv/awesome/www;

access\_log /srv/awesome/log/access\_log;

error\_log /srv/awesome/log/error\_log;

*# server\_name awesome.liaoxuefeng.com; # 配置域名*

*# 处理静态文件/favicon.ico:*

location /favicon.ico {

root /srv/awesome/www;

}

*# 处理静态资源:*

location ~ ^\/**static**\/.\*$ {

root /srv/awesome/www;

}

*# 动态请求转发到9000端口:*

location / {

proxy\_pass http:*//127.0.0.1:9000;*

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header Host $host;

proxy\_set\_header X-Forwarded-**For** $proxy\_add\_x\_forwarded\_for;

}

}

然后在/etc/nginx/sites-enabled/目录下创建软链接：

$ pwd

/etc/nginx/sites-enabled

$ sudo ln -s /etc/nginx/sites-available/awesome .

让Nginx重新加载配置文件，不出意外，我们的awesome-python3-webapp应该正常运行：

$ su**do** /etc/init.d/nginx reload

如果有任何错误，都可以在/srv/awesome/log下查找Nginx和App本身的log。如果Supervisor启动时报错，可以在/var/log/supervisor下查看Supervisor的log。

如果一切顺利，你可以在浏览器中访问Linux服务器上的awesome-python3-webapp了：



如果在开发环境更新了代码，只需要在命令行执行：

$ fab build

$ fab deploy

自动部署完成！刷新浏览器就可以看到服务器代码更新后的效果。

### 友情链接

嫌国外网速慢的童鞋请移步网易和搜狐的镜像站点：

<http://mirrors.163.com/>

<http://mirrors.sohu.com/>

### 参考源码

[day-15](https://github.com/michaelliao/awesome-python3-webapp/tree/day-15)

### 感觉本站内容不错，读后有收获？

#### Day 16 - 编写移动App

阅读: 66408

网站部署上线后，还缺点啥呢？

在移动互联网浪潮席卷而来的今天，一个网站没有上线移动App，出门根本不好意思跟人打招呼。

所以，awesome-python3-webapp必须得有一个移动App版本！

### 开发iPhone版本

我们首先来看看如何开发iPhone App。前置条件：一台Mac电脑，安装XCode和最新的iOS SDK。

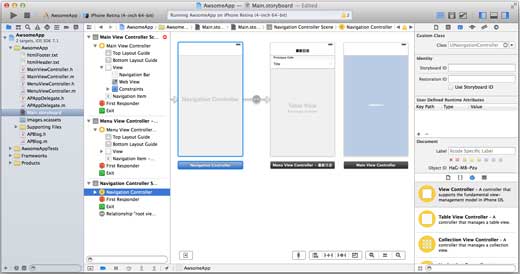
在使用MVVM编写前端页面时，我们就能感受到，用REST API封装网站后台的功能，不但能清晰地分离前端页面和后台逻辑，现在这个好处更加明显，移动App也可以通过REST API从后端拿到数据。

我们来设计一个简化版的iPhone App，包含两个屏幕：列出最新日志和阅读日志的详细内容：



只需要调用API：/api/blogs。

在XCode中完成App编写：



由于我们的教程是Python，关于如何开发iOS，请移步[Develop Apps for iOS](https://developer.apple.com/technologies/ios/)。

[点击下载iOS App源码](https://github.com/michaelliao/awesome-python3-webapp/tree/day-16/ios)。

如何编写Android App？这个当成作业了。

### 参考源码

[day-16](https://github.com/michaelliao/awesome-python3-webapp/tree/day-16)

### 感觉本站内容不错，读后有收获？

#### FAQ

阅读: 236

### 常见问题

本节列出常见的一些问题。

### 如何获取当前路径

当前路径可以用'.'表示，再用os.path.abspath()将其转换为绝对路径：

*# -\*- coding:utf-8 -\*-*

*# test.py*

**import** os

print(os.path.abspath('.'))

运行结果：

$ python3 test.py

/Users/michael/workspace/testing

### 如何获取当前模块的文件名

可以通过特殊变量\_\_file\_\_获取：

*# -\*- coding:utf-8 -\*-*

*# test.py*

**print**(\_\_file\_\_)

输出：

$ python3 test.py

test.py

### 如何获取命令行参数

可以通过sys模块的argv获取：

*# -\*- coding:utf-8 -\*-*

*# test.py*

**import** sys

print(sys.argv)

输出：

$ python3 test.py -a -s "Hello world"

['test.py', '-a', '-s', 'Hello world']

argv的第一个元素永远是命令行执行的.py文件名。

### 如何获取当前Python命令的可执行文件路径

sys模块的executable变量就是Python命令可执行文件的路径：

*# -\*- coding:utf-8 -\*-*

*# test.py*

**import** sys

print(sys.executable)

在Mac下的结果：

$ python3 test.py

/usr/local/opt/python3/bin/python3.4

### 感觉本站内容不错，读后有收获？

#### 期末总结

阅读: 12754

终于到了期末总结的时刻了！

经过一段时间的学习，相信你对Python已经初步掌握。一开始，可能觉得Python上手很容易，可是越往后学，会越困难，有的时候，发现理解不了代码，这时，不妨停下来思考一下，先把概念搞清楚，代码自然就明白了。

Python非常适合初学者用来进入计算机编程领域。Python属于非常高级的语言，掌握了这门高级语言，就对计算机编程的核心思想——抽象有了初步理解。如果希望继续深入学习计算机编程，可以学习Java、C、JavaScript、Lisp等不同类型的语言，只有多掌握不同领域的语言，有比较才更有收获。



**感觉本站内容不错，读后有收获？**